# Fast Physical Simulation of Virtual Clothing based on Multilevel Approximation Strategies

*James N. Anderson*

**Thesis submitted for the degree of Doctor of Philosophy**


**1998**

# Abstract

One of the primary disadvantages of Internet telepresence implemented using currently available technology is the lack of 'touch and feel' available to users. This problem is particularly acute with regard to Internet retailing, since customers generally prefer to try clothing on before purchasing it; even those customers who buy items via mail order catalogues exhibit a high return rate and do so at considerable cost to the retailer.

A virtual mannequin service, which uses a computer-generated representation of the consumer to 'try on' different items and combinations of virtual clothing, would provide an excellent solution to this problem. The consumer could easily see how various sizes, styles, and combinations of garments would look on their own person yet without leaving the home or office. However, until now the possibility of such a service for telepresence shopping systems has been unrealistic due to the number and complexity of calculations required for the modelling of physical clothing items.

This thesis presents a full account of the FIGMENT scheme (Fast Implementation Garment Modelling environmENT) which incorporates a four-point strategy (a simplified physical model, collision volume approximation, progressive meshes and a hybrid rendering algorithm) acting at multiple levels in the modelling process to reduce the quantity and complexity of the computations involved, bringing modelling times from the realm of hours to minutes and seconds whilst maintaining an acceptable level of accuracy and fidelity in the results. The physical model permits garment models obtained by various methods to be used in simulations, incorporates alternative methods of force computation to allow a range of speed-accuracy levels, and provides a robust basis for the other aspects of the scheme. The two methods of collision volume approximation presented enable collision handling in $O(n)$ time rather than the $O(n\log n)$ time of optimised polygon-to-polygon detection methods whilst providing other advantages germane to the modelling process. The further development and employment of progressive mesh algorithms permits an additional increase in modelling rates without loss of fidelity. Finally, the use of a hybrid rendering algorithm which combines depth-buffering and depth-sorting techniques effectively masks the minor visual discrepancies introduced by the other points of the scheme and enables the use of multilayered complex garments without resorting to cloth-to-cloth collision methods (both of which would require considerable additional computation to otherwise achieve), whilst only marginally affecting modelling rates. When fully implemented, the FIGMENT scheme can reduce modelling times by a factor of 80 in typical cases.

The aim of the thesis is to detail the design principles and the algorithms which together comprise the FIGMENT scheme and to demonstrate by way of example and user tests the benefits typically afforded by implementing a virtual mannequin service based on the scheme.

The contribution of this thesis is therefore to present a set of novel approaches to modelling clothing which would together allow the implementation of a virtual mannequin service using the levels of hardware and software technology which presently exist in the majority of computer-equipped homes and offices, yet without hindering the usability of such a service.

# Acknowledgments

I would like to take this opportunity to thank the following people for their contribution, in one way or another, to the completion of this work

Firstly, I should like to thank my supervisor, Prof Mervyn Jack, for his friendly guidance and support (both academic and financial) during the past three years, and for providing just the right level of interaction with, and oversight of, my work throughout that period. My thanks also go to my second supervisor, Dr Gordon Povey, for his helpful comments at key stages in the development of that work.

Dr John Foster and Dr Fergus McInnes, at the Centre for Communication Interface Research, also deserve my gratitude for their help with the design and execution of the experiment described in Chapter 7. Further thanks go to John and Dr Steve Love (also at the CCIR) for providing background information and technical details regarding the Likert questionnaire format.

On a different note, I thank my colleagues at the CCIR, most notably Iain McKay, for providing friendship and regularly brightening the day with a touch of humour from those early undergraduate project days, many moons ago, until now.

I'm immensely grateful to my parents, David and Judith Anderson, without whose love, encouragement, perseverance, prayers and hard-earned cash I would surely not be where I am today.

Finally, I must express my heartfelt thanks to my dear wife, Catriona, for her saintly patience and loving support (particularly during those times when I really wondered whether I would ever reach this point) and to my Father God by whose grace alone I am what I am (1 Corinthians 15:10).

# Contents

# List of Figures

# List of Tables

# Glossary

**axis-aligned bounding box** : the smallest box in three-dimensional space with edges parallel to the major axes which entirely surrounds an object (Chapters 3 and 4)

**capsule** : a geometric structure used for collision volume approximation (Chapter 3)

**edge-collapse** : a geometric transformation used for polygonal mesh simplification (Chapter 5)

**equilibrium dimensions** : the dimensions of an undeformed ('at rest') triangular cloth section

**HTML** : Hyper-Text Markup Language

**Java** : object-oriented, network-based, platform-independent programming language

**joint** : the shared edge of two adjacent cloth sections between which bending forces may act (Chapter 2)

**node** : the mass-point component of a cloth mesh (Chapter 2)

**OBB** : oriented bounding box; a rectangular bounding box at an arbitrary orientation in 3D space

**octree** : hierarchical tree structure with up to eight children per parent (Appendix A)

**Poisson coefficient** : measurement of a material's lateral contraction when elongated (also called *Poisson's ratio*)

**radial depth** : the shortest distance from a point on the surface of a polygonal object to a specified lateral axis (Chapter 4)

**section** : the triangular division of a cloth surface defined between three nodes (Chapter 2)

**vertex-split** : the reverse transformation to an edge-collapse (Chapter 5)

**VRML** : Virtual Reality Modelling Language

**Young's modulus** : the constant of proportionality between strain and stress in elastic deformation (also called the *modulus of elasticity*)

# Chapter 1

# Introduction

## 1.1 Introduction

The advent of global networking has radically changed the way people live and work—and will continue to do so. The World Wide Web, whilst not quite matching God's unique prerogative of having "the whole world in His hands", at least permits a respectable proportion of the world to be squeezed into offices and front rooms. Combine this high-speed global interconnectivity with the phenomenal rate at which computer hardware is developing, increasing in performance whilst decreasing in price, along with the accompanying software to maximise the potential of that hardware, and the possibilities for applications in the areas of education, entertainment, manufacturing, commerce and research are seemingly endless.

Human nature and western society being what they are, it has been of little surprise to observe that the first applications of these new technologies have focused most visibly on the two closely-related realms of entertainment and consumerism. For the former, the development of advanced multimedia technologies (fast 3D graphics hardware in particular) has not only allowed the previously existing stock of game formats to be rejuvenated but has introduced a whole new set of genres altogether, while high-bandwidth low-latency networks have opened the door to (literally) new dimensions in the form of multi-player scenarios. For the latter, however, the real potential of such technologies has only just begun to be exploited, as is explained in this thesis.

## 1.2 Telepresence shopping and its limitations

'Telepresence' refers to the electronically simulated presence of a user within a remote environment by means of advanced telecommunications technology. Telepresence shopping is arguably the ultimate goal of online consumer services; it offers the possibility of 'making a good buy without saying a good-bye'. Of course, in the broadest possible sense, telepresence shopping has been around ever since retailers began using glossy paper catalogues, allowing customers to order the contents over the telephone. Yet this hardly amounts to 'simulated presence' in any significant manner. A true telepresence service needs to allow the consumer to perform, to the greatest extent possible, all of the actions that he or she might usually perform when shopping in 'real-life'. These actions include examining products, trying them out (or on, if appropriate), obtaining further information, making comparisons with other retailers, discussing the pros and cons with companions, browsing, relaxing and being entertained. Furthermore, an effective telepresence service needs to detach users from their actual settings, giving the impression that they are 'present' in another 'location' altogether. This will involve the simulation of the sensory experiences appropriate to that 'location'; primarily visual and auditory experiences, possibly tactile, and even (someday) olfactory and gustatory. Naturally, one of the great advantages of telepresence shopping is that these experiences need not be the same experiences that are to be found when strolling down to the local supermarket, but could be a considerably more pleasing and personalised set of experiences altogether.

The type of multisensory experience that the ultimate telepresence service might provide is still beyond the reach of current technologies, however rapidly they may have advanced over recent years. Three-dimensional visual and audio simulation is certainly coming of age, but attempts to stimulate the remaining senses are currently crude and clumsy affairs. 'Virtual reality', as it is propounded today, must be considered solely in terms of the audio-visual.

So telepresence shopping, in the narrow sense, has not yet arrived. However, the absence of a five-fold sensory experience should not prevent the implementation of

telepresence retailing services in the meantime. After all, it is not essential for a consumer to 'feel' a video recorder in order to make an informed and comfortable decision about its functionality, performance and style. The consumer does not need to 'smell' or 'taste' a compact disc before making a purchase. Thus, for a considerable proportion of products which might be offered by an online store, provision for the two primary senses are sufficient at the present time.

However, in terms of the possibilities of selling clothing via a telepresence shopping service, the situation and requirements are markedly different. Although glossy paper catalogues have enjoyed high levels of success, this does not come without a cost. Some 30% of purchased clothing items are reportedly returned and a considerable proportion of customers will order two or more sizes of item, intending to return all but one (if not all). For good reason, too. The dress may look marvelous on the model in the photograph, but that gives little indication to the individual customers of how it will actually look on *them*. Photographs—whether they appear in a paper catalogue or on a computer screen—can only provide so much information about a product, and that information is too limited for most customers to be comfortable with any form of 'armchair shopping'.

What sort of scenario can be envisaged, then, where the customer would be prepared to buy an item of clothing from home?

## 1.3 The Virtual Mannequin

'Virtual mannequin' technology provides an answer to the question posed above. Such technology involves the implementation of an accurate *computer simulation* of the customer, reflecting their physical dimensions and appearance. With the appropriate software, this computer-generated representation can be used to 'try on' three-dimensional models of clothing items and rendered with potentially photographic quality on the screen of the customer's home computer.

A 'virtual mannequin' service developed along these lines would be enough to satisfy most customers' requirements, but the application of computer technology need not end there. At the click of a mouse button, the customer could see the same

item in a different style or the next size up or down fitted to their virtual mannequin. For the more demanding telepresence customer, a made-to-fit *virtual* garment is the logical step prior to ordering a tailor-made garment itself.

The virtual mannequin service can also retain the details of all of the customer's previously purchased clothes and can quickly render an image of the customer wearing the potential purchase with any combination of those clothes, not to mention jewellery, make-up or hair-styles. Furthermore, combinations of clothes from a number of different shops may be tried before opting to buy any of them.

One of the primary advantages of the virtual mannequin is the ability to 'try on' and buy clothes without leaving home or office. Yet the same network which brings the mannequin into one particular customer's home or office can with equal ease bring that same mannequin simultaneously to *other* homes and offices. The potential for 'collaborative shopping' applies to clothes as much as to any other product.

A number of major clothing retailers with online stores have recognized the competitive edge that such a service would give them.[1] But is it really feasible with today's Internet technology?

## 1.4 Modelling clothing

Creating a three-dimensional computer-generated representation of a customer's body presents no insuperable difficulty. Full-body scanners are readily available, and only one trip to the 'scanning station' would be required. However, such a highly-detailed representation is not necessary for a usable mannequin service. With the aid of a tape measure, customers could provide sufficient information from the comfort of their own homes to produce personal mannequins accurate enough for the purposes of modelling clothing. Skin colour may be easily adjusted and a few images of the customer's head (obtained either via video-conferencing hardware or from passport photographs) would allow the mannequin to truly become a 'digital twin'.

[1] For example, the company GAP (**http://www.gap.com**) has considered the idea of a virtual fitting room and (at the time of writing) their online catalogue features a simple two-dimensional interactive 'dressing room' which allows customers to experiment with various combinations of garments.

The mannequin customisation phase of the service is perfectly feasible with currently technology.

However, the one major hurdle which prevents present-day implementation of 'virtual mannequin' services is the number and complexity of the computations required for the physical simulation of cloth. The fabric of the garments should be seen to hang, bend, fold and crease in the same way as the real fabric would, interacting appropriately with the solid form of the mannequin body so that the 'fitting' of the clothing is representative and informative. Since the calculations required for such physical simulation are non-trivial and involve interactions between sets of thousands of discrete elements, the computational requirement is intense, taking hours if not days to produce frames for mere seconds of animation. The results can be most impressive, as a number of recent computer-generated films have demonstrated,[2] but the time factors involved mean that simply implementing the same modelling algorithms within a near real-time virtual mannequin service for the Internet is out of the question.

Nevertheless, although the computational requirement may seem prohibitive, there are alternative approaches beyond patiently waiting for the next generation (or later) of floating-point co-processors and then for the price to fall below the bank balance of the average clothes shopper.

## 1.5 The FIGMENT scheme

In order to simulate the clothing of a virtual mannequin on a customer's home or office PC at interactive rates, some compromise must be made by trading accuracy, or level of detail, against speed of computation. This loss of accuracy need not significantly affect the usability and advantages of the virtual mannequin service, since such a service is not intended to match or replace currently available multimedia technology but to supplement it. High-quality photographic detail can already be provided by two-dimensional images and movies. In contrast, the

[2] See in particular the work of MIRALab (**http://miralabwww.unige.ch**) at the University of Geneva.

emphasis within the mannequin service will be that of *context-based visualisation*. Rather than being restricted to a static and impersonal view of models wearing the clothing, users will be able to see those garments in the context of *(1)* bodies with their own physical dimensions, *(2)* their own skin colour, hair style, *etc.*, *(3)* different sizes, colours and styles, *(4)* other garments and *(5)* different environmental conditions, *e.g.* lighting. The advantages afforded by such technology, even if failing to deliver the highest level of accuracy with photographic-quality detail, will act as a useful and perhaps indispensable supplement to non-interactive two-dimensional media for the Internet shopper.

The simplest way to effect this compromise—speed traded against accuracy and visual fidelity—is to use reduced-complexity (more discrete) models for both the garment and the mannequin. There are limits, however, to the extent to which this can be done before the usability of the service is lost. A better approach would be to combine the use of partially simplified models with optimised algorithms for simulating the dynamics and appearance of the garments as they interact with the surface of the mannequin. In fact, optimisation and simplification can be applied at multiple levels in the modelling process—from the basic dynamics of the physical elements of the fabric to the final image rendering—with each aspect contributing to the overall speed gain. In this way, a dramatic reduction in simulation time can be achieved while avoiding the prohibitive loss of fidelity resulting from the simplistic former approach.

The FIGMENT scheme (Fast Implementation Garment Modelling environmENT) described in this thesis takes the latter approach in order to allow for a usable implementation of a virtual mannequin service. The scheme incorporates a four-point approach towards optimising the necessary computations and reducing the overall modelling times whilst maintaining an acceptable level of accuracy and fidelity in the rendered results. Firstly, a simplified physical model is used to represent the dynamic interactions between discrete sections of cloth and their surrounding environment. Secondly, collision volume approximation methods are used to represent the solid surface of the mannequin, significantly reducing this time-consuming phase of physical simulation. Thirdly, all or part of the garment models

are replaced with so-called 'progressive mesh' representations, speeding up the early stages of the simulation. Finally, a hybrid rendering algorithm is used to ensure that multiple layers of clothing can be rendered without appearing to penetrate one another or the surface of the mannequin, despite minor collision handling imprecision and in the absence of cloth self-collision algorithms. The four points of the FIGMENT scheme, although self-contained, are mutually supportive of one another and when implemented together allow for a superior service than would be achieved if used separately. The aim of this thesis is to detail the design principles and algorithms which together comprise the FIGMENT scheme and to demonstrate by way of example and user tests the benefits typically afforded by implementing the scheme.

The FIGMENT scheme represents a significant contribution to the areas of interactive computer simulation in general and multimedia consumer services in particular. Without such a scheme, the implementation of a usable virtual mannequin must await significant future developments in the processing capabilities of desktop hardware. The FIGMENT algorithms have been developed in an application-specific context and therefore take advantage of the optimizations presented by that context. Yet despite that specificity, aspects of the scheme may well prove profitable in applications beyond that of online shopping such as virtual sports or virtual communities.

## 1.6 Chapter summaries

The four points of the FIGMENT scheme are dealt with in Chapters 2 through 6. Chapter 2 reviews various approaches to simulating the physical dynamics of fabric and discusses the advantages and disadvantages of each with respect to the present application. The chapter argues for the adoption of a particular physical model before specifying in detail the optimised algorithms developed in order to implement that model within a FIGMENT-based application. Issues regarding instability and inaccuracy at larger simulation time-steps are discussed and a solution offered.

Finally, the speed gain, accuracy and visual results obtained via the FIGMENT physical model are assessed by examining some typical modelling simulations.

In Chapter 3, the problems of collision detection and response are examined. Various approaches to collision handling, in both non-real-time and real-time systems, are reviewed and found to be inadequate for the present application in a number of crucial areas. An alternative method of collision handling—collision volume approximation—is presented as a suitable solution. One form of collision volume approximation, the 'capsule' method, is then introduced before detailing the algorithms required for collision detection and response according to this method. The chapter concludes by providing examples of the speed gains afforded by the 'capsule' method, the cost with regard to accuracy, and the visual results obtained, all with respect to a representative algorithm for collision detection widely used in non-real-time applications which involve the physical modelling of cloth.

Chapter 4 continues the discussion of collision handling algorithms by providing an alternative method of collision volume approximation, the 'radial depth' method, which provides a greater degree of accuracy with respect to the 'capsule' method but with a slightly greater computational requirement. The algorithms required for collision detection and response according to the 'radial depth' method are presented in detail. As in the previous chapter, this chapter concludes by using the same example simulations to assess the speed gains afforded by the 'radial depth' method, the cost with regard to accuracy, and the visual results obtained, all with respect to both the comparison collision detection algorithm and the 'capsule' method.

The third point of the FIGMENT scheme is discussed in Chapter 5, beginning with an assessment of the advantages and problems introduced by using reduced-complexity cloth meshes for garment modelling. The main methods of model simplification are reviewed, all of which are intended for non-deformable objects, and found to be unsuitable for various reasons in their present form. It is argued, however, that one simplification technique may be modified and developed to allow for the use of lower-complexity cloth meshes in a way which permits significant reductions in simulation time yet maintains the visual fidelity of the final results by progressively restoring the original level of complexity as the simulation proceeds.

The chapter details the algorithms used to obtain optimal lower-complexity cloth meshes as well as the corresponding algorithms required in order to use the meshes for simulations in which not only the geometry but the 'physical' attributes and dynamic deformation of the meshes must be correctly attended to. Finally, the speed gains afforded by this component of the FIGMENT scheme is demonstrated by comparing some typical modelling simulations; the accuracy cost and visual results obtained are also assessed.

Chapter 6 covers the final point of the FIGMENT scheme: the hybrid rendering algorithm. After reviewing the various rendering algorithms commonly implemented by computer graphics libraries and discussing the limitations of each algorithm with respect to garment modelling, the chapter argues that a modified algorithm which combines the approaches of the two most widely used real-time rendering algorithms could overcome these limitations. The form and implementation of such an algorithm are described in detail along with solutions to two major problems encountered when using the algorithm in practice. The remaining limitations of the hybrid rendering algorithm in its present form and potential solutions are discussed before concluding the chapter with an examination of the computational cost and visual results obtained when using the algorithm in practice.

In Chapter 7, the focus turns from the theoretical to the practical aspect of the FIGMENT scheme by considering the implementation of a FIGMENT-based mannequin service and user responses to such a service. The majority of this chapter describes a user trial performed with a simple implementation of a mannequin service, the aim of which was to establish experimentally that the goal of the FIGMENT scheme has been achieved: modelling clothing at interactive rates with no significant loss of fidelity or accuracy. The chapter also discusses the possibility of a VRML-based mannequin service using the FIGMENT scheme.

Chapter 8 draws overall conclusions from both the theoretical and practical aspects of the FIGMENT scheme, discusses its present limitations and suggests directions for future work.

## 1.7 Summary

A 'virtual mannequin' would provide an ideal solution to the problems faced by clothing retailers who wish to provide online 'telepresence' shopping services. Although the physical modelling of clothes involves a considerable computational requirement, a combination of optimised algorithms and approximating techniques acting at various levels in the modelling process can allow for the implementation of a mannequin service which provides results rapidly with minimal detriment to the fidelity and accuracy of the visual results. The FIGMENT scheme offers an integrated four-point approach in order to achieve this end: a simplified physical model, collision volume approximation, 'progressive mesh' representations, and a hybrid rendering algorithm.

# Chapter 2

# The Physical Model

## 2.1 Introduction

This chapter provides a detailed account of the first point of the FIGMENT scheme—the physical model used to simulate the dynamic behaviour of the cloth sections which comprise the modelled garments. As with all four points of the FIGMENT scheme, the emphasis in formulating the physical model is that of speed optimization whilst maintaining acceptable levels of fidelity in the visible results. With these considerations, the various approaches towards modelling fabric which have been offered in the literature are discussed and reviewed for suitability, subsequently giving reasons for the adoption of the particular model in question.

Having chosen a fundamental approach to modelling, the actual calculations required in order to compute the forces (both external and internal to the fabric) are given in detail. Following this, experimental data comparing the computation speed of 'standard' and 'fast' algorithms are given.

Increasing the time divisions for iterative simulations, while advantageously reducing the overall simulation time, can introduce instability into the model which can often prove fatal to the modelling process. The chapter therefore explains the approach adopted within the FIGMENT scheme towards countering such instability. Then, after detailing a suitable method of assessing the relative accuracy of a particular cloth modelling simulation, a number of typical modelling sessions with a range of controlling parameters are compared and the results discussed.

Finally, conclusions are drawn regarding the strengths and weaknesses of the FIGMENT physical model and its application within a virtual mannequin service.

## 2.2 Fabric simulation techniques

The literature regarding physically-based cloth modelling over the last ten years can be usefully classed into four approaches to dynamic fabric simulation. Firstly, Terzopoulos and Fleischer (1988) describe a general model for animating non-rigid objects which requires, in practice, the formulation of energy functions for the nodes of a discrete mesh which determine the dynamics of those nodes over time. The advantages of the technique include its application to a wide range of deformable materials and its inclusion of both elastic and inelastic phenomena, resulting in realistic simulations. In addition, the integrative nature of the computation minimizes the overall error incurred as the simulation progresses. Its major disadvantage, however, is that of the complex calculations involved (multi-dimensional linear equation solutions) and the resulting high computational intensity.

Secondly, Breen, House and Wozny (1994a, 1994b) have developed a particle-based model for simulating draping behaviours in woven cloth. The technique also requires the formulation of energy functions representing both the elastic interaction between the nodes of discrete quadrilateral meshes (repelling, stretching, bending and trellising) and also environmental forces, such as gravity. The simulation proceeds by performing energy minimization calculations in order to determine the dynamic behaviour of the cloth. As with the Terzopolous-Fleisher approach, the computation is integrative in nature and is similarly computationally intensive. It should be noted that there is also some difficulty in directly relating the physical parameters of the cloth used in this method to those most commonly specified with respect to elastic materials (*e.g.* Young's modulus, Poisson coefficient, density and thickness). The authors make use of so-called Kabawata testing (Kabawata, 1980) to obtain the necessary parameters for various types of actual fabric.

A third method is that employed by the MIRALab team at the University of Geneva. Volino, Courchesne and Magnenat Thalmann (1995) have presented a

physical model based on Newtonian mechanics and acting on irregular meshes of triangles. In practice, the simulation proceeds iteratively by computing, during each time-step, both the elastic and shearing strains on individual triangles and the bending strain between adjacent triangles. The forces acting on the edges and corners of each triangle are thus calculated, and the overall change in dynamics (position, velocity and acceleration) is estimated over each time-step using the second-order (midpoint) Euler-Cromer numerical method (Kreyszig, 1988). The model also implements linear viscoelastic response and plastic behaviour for large deformations. The advantages of this approach include the relatively low computation requirement (compared to the previously mentioned methods), the ability to use irregular cloth meshes, and the accuracy of the simulation which takes into account standard physical parameters such as Young's modulus and Poisson coefficient. In addition, the iterative nature of the calculations allows, in contrast to the analytic methods, arbitrary and complex collision and environmental forces to act on the cloth as well as internal physical forces. More recently, Volino and Magnenat Thalmann (1997) have developed the model to more closely resemble the spring-mass system (see below), avoiding the need to perform geometrical calculations in local coordinates whilst maintaining, to a high degree, the physical accuracy of the original model. They have also adopted the more accurate (but more complex) Runge-Kutta iterative integration method (Kreyszig, 1988) in order to, amongst other things, maintain stability when using large time-steps.

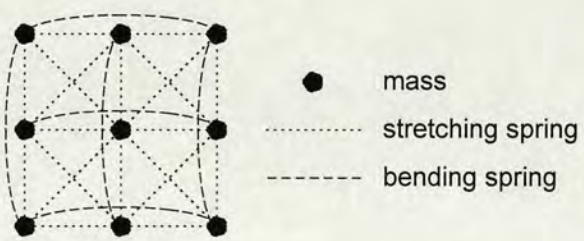



Figure 2.1: Mass-spring system for physical cloth model

Finally, Provot (1995) has provided a good example of a mass-spring system, which is generally the simplest and fastest method for modelling cloth. The mesh is considered as a distribution of mass particles connected by damped springs. These

springs act between pairs of adjacent masses to simulate tension and shearing forces and between non-adjacent masses to simulate bending forces (Fig. 2.1). The majority of such models require mesh regularity in order to obtain accurate results—in any case, the overall results are generally less accurate than those of the first three methods. Provot (1995) uses a regular quadrilateral mesh. The clearest advantage of the mass-spring method over those outlined above is the simplicity and speed of computation; the main disadvantages are those of accuracy and mesh regularity. The additional problem of accurately relating spring tension constants to the standard physical parameters should also be noted.

## 2.3 The FIGMENT physical model

The choice of a basic physical model for the FIGMENT scheme needs to take the following specifications into account:

- The computation required should be minimal whilst maintaining significant accuracy in the results.
- The model should allow for additional complex, dynamic collision and environmental effects (such as air resistance, surface friction, and wind).
- The model should allow for the non-regular meshes required by arbitrarily shaped cloth sections.
- A model based on triangular, rather than quadrilateral, elements is preferable. The latter exhibit surface ambiguity which can, in cases with relatively large discretization of surfaces, result in rendering and collision inaccuracies.

On the basis of the first and second points, both the Terzopolous-Fleisher model and the Breen-Wozny model must be immediately rejected. The Provot model, as it stands, is unsuitable due to its regular quadrilateral mesh requirement. An adaptation using triangular elements would be possible, although the issues of regularity and accuracy would still remain.

For these reasons, it seems clear that the Volino-Courchesne-Thalmann model offers the most suitable basis for the FIGMENT scheme, combining the advantages

of irregular triangular meshes, the use of standard physical parameters, computational simplicity and fidelity of results.

The basic FIGMENT physical model therefore consists of clothing items represented by irregular triangle meshes, the dynamics of which are computed by considering the effects of Newtonian mechanics (internal and external forces) acting on their discrete elements. The mesh is considered in practice to be a mass-particle system; the mass-particles (hereafter, **nodes**) occur at the corners of triangular discretizations (hereafter, **sections**) which are hinged at adjoined edges (hereafter, **joints**). The mass of each section is equally distributed to its three nodes according to its area, thickness and density.

There are essentially six types of forces acting on each triangular section (specifically, at its corner nodes): gravitational force, tensional forces, flexional (bending) forces, frictional forces, air resistance, and wind effects.

Gravitational force

The gravitational force acting on any one node is taken as the product of its mass and the gravitational acceleration constant.

Tensional forces

The tensional forces acting on any one section depend on the deformation of that section from its equilibrium (at-rest) state. By calculating the normal stress, $\sigma$, and the shear stress, $\tau$, present within the section, the resultant forces acting perpendicular to each edge can be computed. The calculations are most straightforwardly performed by considering the geometry of the section in terms of its own two-dimensional local coordinate system such that the longest side of the section acts as the $x$-axis (Fig. 2.2). In this way, then, the triangle is fully defined according to the length of its longest side and the coordinates of the opposite corner, and these values are precomputed before simulation for the equilibrium state of each section ($l_0$, $x_0$ and $y_0$).

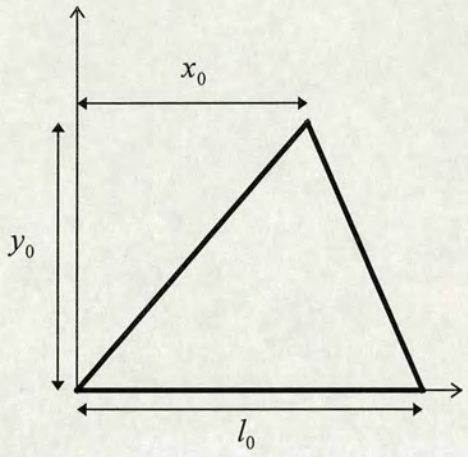



Figure 2.2: Equilibrium state of section in two dimensions

In order to calculate these values for the *deformed* section in three-dimensional space (with edge vectors $\overline{v}_{AB}$, $\overline{v}_{BC}$, $\overline{v}_{CA}$, and normal vector $\overline{n}$) the following preliminary vectors and geometrical ratios are computed (Fig. 2.3):

$$\overline{u}_{AB} = \frac{\overline{v}_{AB}}{|\overline{v}_{AB}|} \qquad \overline{u}_{BC} = \frac{\overline{v}_{BC}}{|\overline{v}_{BC}|} \qquad \overline{u}_{CA} = \frac{\overline{v}_{CA}}{|\overline{v}_{CA}|}$$

$$\cos\phi = -\overline{u}_{AB} \cdot \overline{u}_{BC} \quad \sin\phi = \sqrt{1-\cos^2\phi}$$

$$\cos\theta = -\overline{u}_{AB} \cdot \overline{u}_{CA} \quad \sin\theta = \sqrt{1-\cos^2\theta}$$

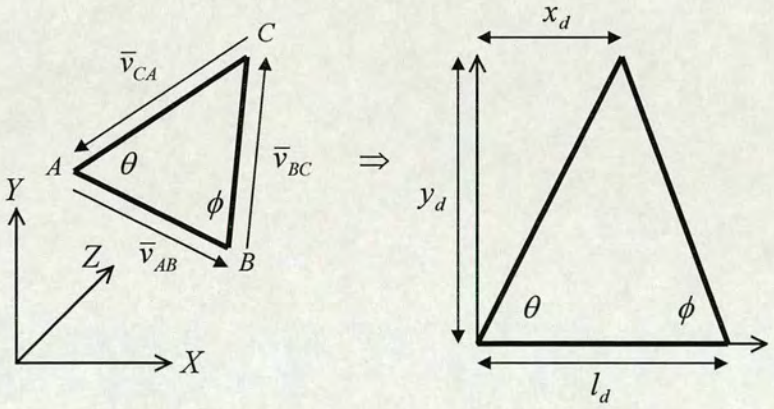



Figure 2.3: Calculating the dimensions of a deformed section

The three defining values ($l_d$, $x_d$ and $y_d$) are then calculated thus:

$$l_d = |\bar{v}_{AB}| \qquad x_d = |\bar{v}_{CA}|\cos\theta \qquad y_d = |\bar{v}_{CA}|\sin\theta$$

Once these values are obtained, the elastic strain magnitudes (normal strain along each axis and shear strain) in the deformed section are as follows (Hibbeler, 1994):

$$\varepsilon_x = \frac{l_d}{l_0} - 1 \qquad \varepsilon_y = \frac{y_d}{y_0} - 1 \qquad \gamma_{xy} = \frac{x_d - x_0(\varepsilon_x + 1)}{y_d}$$

Finally, the elastic stress magnitudes (see Fig 2.4) are obtained as follows, where $E$ and $\nu$ correspond, respectively, to the Young's modulus and Poisson coefficient of the cloth material (Hibbeler, 1994):

$$\sigma_x = E\frac{\varepsilon_x + \nu\varepsilon_y}{1-\nu^2} \qquad \sigma_y = E\frac{\varepsilon_y + \nu\varepsilon_x}{1-\nu^2} \qquad \tau_{xy} = E\frac{\gamma}{2(1+\nu)}$$

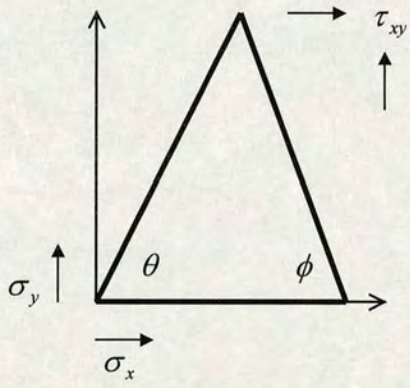



Figure 2.4: Elastic stress magnitudes in deformed section

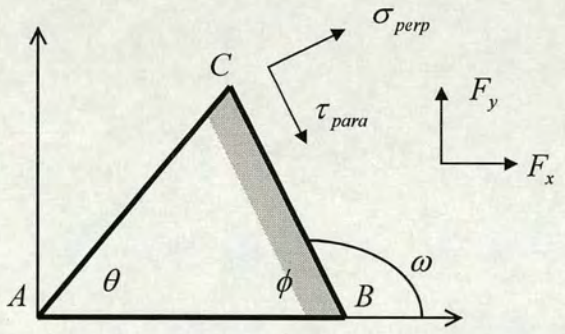



Figure 2.5: Resultant forces acting on edge of section

In order to compute the resultant forces acting on each individual edge (Fig. 2.5), the normal stress perpendicular to the edge, $\sigma_{perp}$, and the shear stress parallel to the edge, $\tau_{para}$, must be calculated. For an edge at an angle $\omega$ from the $x$-axis (in the local coordinate system) those stress values are given by the following equations (Hibbeler, 1994):

$$\sigma_{perp} = \frac{\sigma_x + \sigma_y}{2} - \frac{\sigma_x - \sigma_y}{2}\cos 2\omega - \tau_{xy}\sin 2\omega \tag{2.1}$$

$$\tau_{para} = -\frac{\sigma_x - \sigma_y}{2}\sin 2\omega + \tau_{xy}\cos 2\omega \tag{2.2}$$

From these values, the perpendicular and parallel components of force acting on the edge can be calculated (where $A$ is the cross-sectional area of the edge) and thus the $x$- and $y$-components of force, $F_x$ and $F_y$:

$$F_x = \left(-\tau_{para}\cos\omega + \sigma_{perp}\sin\omega\right)\cdot A \tag{2.3a}$$

$$F_y = \left(-\tau_{para}\sin\omega - \sigma_{perp}\cos\omega\right)\cdot A \tag{2.3b}$$

The values of $\sin\omega$, $\cos\omega$, $\sin 2\omega$ and $\cos 2\omega$ for the three cases corresponding to the three edges of the section can be obtained from the previously computed values of $\sin\phi$, $\cos\phi$, $\sin\theta$ and $\cos\theta$. Thus, the force on each of the three edges can be computed as follows (where $T$ is the thickness of the section) by computing the components of force in each of the two dimensions of the local coordinate plane.

For the edge $AB$, the calculations are trivial, since $\omega = 0$:

$$F_{ABx} = -\tau_{para}\cdot T\cdot\left|\bar{v}_{AB}\right| \tag{2.4a}$$

$$F_{ABy} = -\sigma_{perp}\cdot T\cdot\left|\bar{v}_{AB}\right| \tag{2.4b}$$

where

$$\sigma_{perp} = \sigma_y \qquad \tau_{para} = \tau_{xy}$$

For the edge $BC$, substituting $\omega = \pi - \phi$ into (2.1), (2.2), (2.3a) and (2.3b):

$$F_{BCx} = \left(\tau_{para}\cos\phi + \sigma_{perp}\sin\phi\right)\cdot T\cdot\left|\overline{v}_{BC}\right| \tag{2.5a}$$

$$F_{BCy} = \left(-\tau_{para}\sin\phi + \sigma_{perp}\cos\phi\right)\cdot T\cdot\left|\overline{v}_{BC}\right| \tag{2.5b}$$

where

$$\sigma_{perp} = \frac{\sigma_x + \sigma_y}{2} - \frac{\sigma_x - \sigma_y}{2}\left(\cos^2\phi - \sin^2\phi\right) - \tau_{xy}\left(-2\sin\phi\cos\phi\right)$$

$$\tau_{para} = -\frac{\sigma_x - \sigma_y}{2}\left(-2\sin\phi\cos\phi\right) + \tau_{xy}\left(\cos^2\phi - \sin^2\phi\right)$$

For the edge $CA$, substituting $\omega = \pi + \theta$ into (2.1), (2.2), (2.3a) and (2.3b):

$$F_{CAx} = \left(\tau_{para}\cos\theta - \sigma_{perp}\sin\theta\right)\cdot T\cdot\left|\overline{v}_{CA}\right| \tag{2.6a}$$

$$F_{CAy} = \left(\tau_{para}\sin\theta + \sigma_{perp}\cos\theta\right)\cdot T\cdot\left|\overline{v}_{CA}\right| \tag{2.6b}$$

where

$$\sigma_{perp} = \frac{\sigma_x + \sigma_y}{2} - \frac{\sigma_x - \sigma_y}{2}\left(\cos^2\theta - \sin^2\theta\right) - \tau_{xy}\left(2\sin\theta\cos\theta\right)$$

$$\tau_{para} = -\frac{\sigma_x - \sigma_y}{2}\left(2\sin\theta\cos\theta\right) + \tau_{xy}\left(\cos^2\theta - \sin^2\theta\right)$$

The force vectors acting on each edge in three-dimensional space are obtained by multiplying the above force components with unit vectors corresponding to the $x$-axis and $y$-axis of the local coordinate system, and applied distributively to the three nodes at the end of each edge.

Tensional forces (fast method)

It is clear that the standard method described above, although providing accurate results by taking mechanical parameters into full account, requires a considerable amount of computation. The FIGMENT scheme therefore provides a less precise but considerably faster method of calculating forces due to the internal stress within a section.

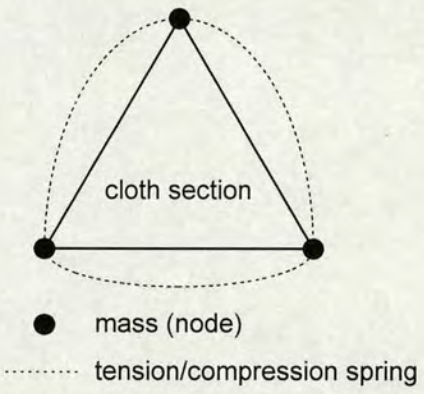



Figure 2.6: Mass-spring model for fast computation of tensional forces

This method simply treats the section as a mass-spring system (Fig. 2.6) in which forces act along each edge of the section according to the elongation or compression of that edge. For each edge $m \rightarrow n$, a constant $K_{mn}$ is precomputed which can be multiplied by the current strain along that edge to provide the magnitude of force:

$$K_{mn} = E \cdot T \cdot \frac{A}{l_{mn}} \tag{2.7}$$

where $E$, $T$, $A$, and $l_{mn}$ are respectively the Young's modulus of the material, the thickness of the section, the equilibrium plane area of the section, and the equilibrium length of the edge. The latter three components provide the average cross-sectional area of the section along the length of the edge. Thus, the magnitude of force acting along each edge is computed to be the product of the spring constant and the strain in the edge:

$$F_{mn} = K_{mn}\left(\frac{|\bar{v}_{mn}|}{l_{mn}} - 1\right) \tag{2.8}$$

Each force vector is then applied (in opposing directions) to the two nodes at either end of the edge. Clearly, this method does not reflect the internal forces as accurately as the previous method, but does provide a considerable speed advantage. The choice between the two methods must be made according to the usability constraints of the particular application in terms of performance, response and fidelity. The results of example simulations which compare the computation of the two methods on various platforms are given below.

Flexional forces

Flexional (bending) forces occur between pairs of joined sections according to the angular displacement between the normal vectors of the sections. To compute the appropriate magnitude of force acting in each case, an estimate of the curvature of the cloth surface at the joint must be made, from which the bending moment can be calculated. Since the analysis involves a discretised surface, force computation requires determination of the *degree of curvature* to be attributed to the angular joint between two sections. The most appropriate method is to specify that a circle (or sphere) with the correct curvature would touch the two sections tangentially at the points equidistant from the joint and the outer nodes (Fig. 2.7). In this way, the 'virtual' surface, as implied by this specification of curvature, will be continuous for either of the typical mesh discretizations: regular right-angled isosceles sections or regular equilateral sections (Fig. 2.8). It should be noted that twist strains are automatically taken into account by means of the additive property of curvature and consideration of Mohr's circle (Volino, 1995).

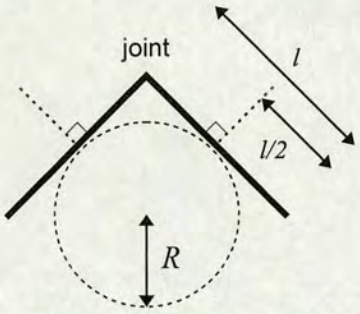
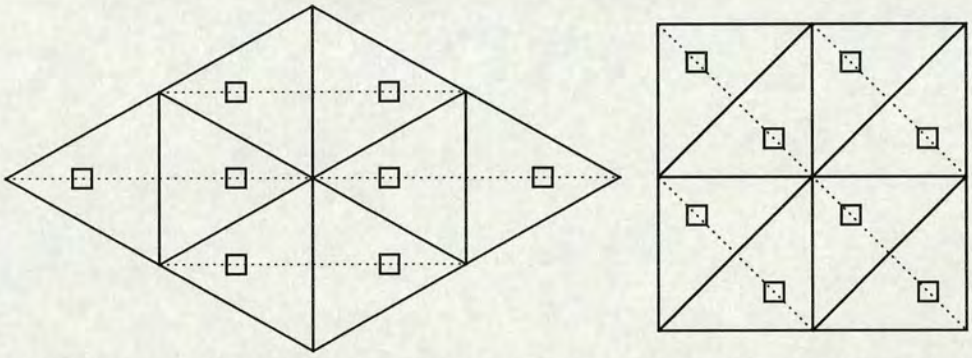



Figure 2.7: Estimation of curvature between jointed sections



**Dotted lines** indicate direction of curvature in cloth mesh; **squares** indicate points at which 'virtual' surface touches plane of sections tangentially; thus, the 'virtual' surface is effectively continuous since squares are aligned perpendicular to direction of curvature.

Figure 2.8: Continuous curvature in regular equilateral sections and regular right-angled isosceles sections

It will be observed that the circle of curvature specified above is only possible if both sections are of equal length, considered from the joint to the outer nodes. The FIGMENT system works on the assumption, therefore, that this is always the case. Considering the ideal case as illustrated in Fig. 2.9, the cosine of the angle between the normal vectors of the sections is obtained initially by calculating the vector dot product of those vectors. If this value exceeds a certain threshold (*e.g.* 0.9999…) then the joint is considered to be unstressed and so no further computation is performed. Otherwise, the distance $d$ between the aforementioned equidistant points is computed as half the distance between the outer nodes of the sections.

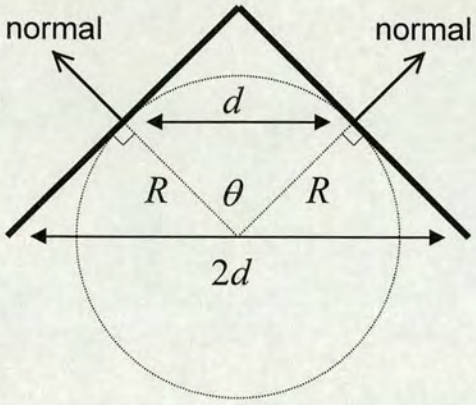



Figure 2.9: Calculation of curvature in ideal case

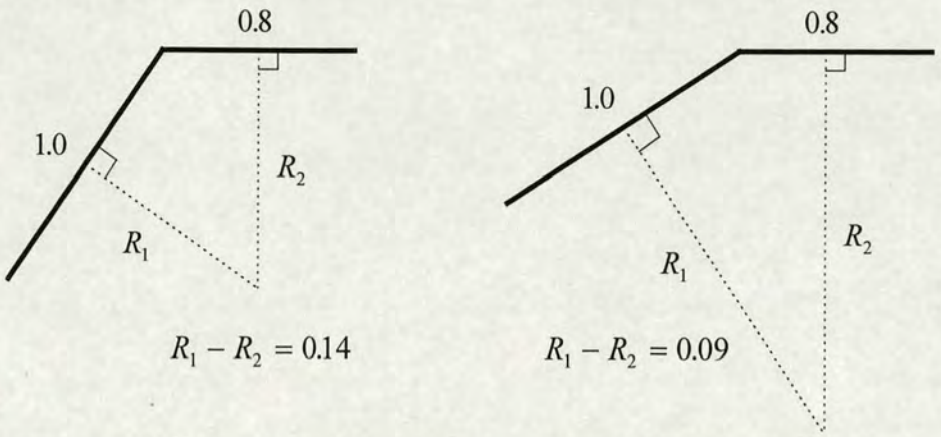
The radius of curvature can be computed from the values of $\cos\theta$ and $d$ as follows:

$$\sin\frac{\theta}{2} = \sqrt{\frac{1}{2}(1-\cos\theta)} \tag{2.9}$$

$$R = \frac{d}{2\sin\frac{\theta}{2}} \tag{2.10}$$

As stated above, the calculations for the ideal case (where the sections are of equal length from joint to outer node) are performed in every case. In *non*-ideal cases, lines drawn perpendicularly from the midpoints of the sections will not meet at a point equidistant from the sections (where that distance would be $R$ in the ideal case). However, by considering the two examples in Fig. 2.10, it can be seen that the discrepancy tends towards zero as the degree of flexure also tends towards zero.[1] The majority of joints within the mesh will not be angled to any great degree; since the flexional forces act to reduce this angle, this method of curvature estimation is quite acceptable in practice.

[1] For the interested reader: When the lengths of the sections are $x$ and $\alpha x$ the discrepancy evaluates to $(x/2)[(1/\sin\theta)(1+\alpha\cos\theta)(1-\cos\theta)-\alpha\sin\theta]$. Hence, when $\alpha=1$, there is no discrepancy.




The resultant error, *i.e.* $R_1$–$R_2$, tends to zero as the curvature tends to zero.

Figure 2.10: Error incurred when making ideal-case assumption

Having calculated a value for the radius of curvature, the bending moment is then computed according to the physical parameters of the fabric as defined previously:

$$M = D \cdot E \cdot \frac{T^3}{12(1-\nu^2)} \cdot \frac{1}{R} \tag{2.11}$$

where $D$ is a numerical constant required to compensate for the fact that the real behaviour of woven fabrics is not that of an ideal continuous solid sheet (for which $D = 1$). The magnitude of the resultant force acting on each outer node is obtained by dividing the bending moment by the distance of that node from the joint; the direction of the force is parallel to the normal vector for that section. However, there is still a residual ambiguity which is not accounted for in the previous calculations, according to two directions in which the bending moment could be acting—whether the cloth is bending 'inward' or 'outward' at the joint.

The most efficient way found to resolve this ambiguity is to compute the vector dot product of the *vector difference* between the normal vectors of the two sections and the *direction vector* between the two outer nodes (Fig. 2.11). The sign of the scalar result indicates in which direction the flexional forces should be applied.

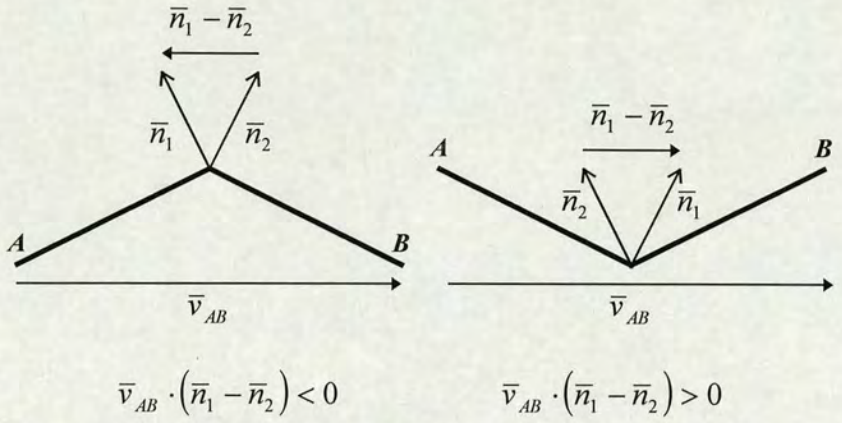



Figure 2.11: Calculating the direction of curvature from section normals

The resultant forces are applied to the two outer nodes and also, in order to balance the net forces on the local system, inversely and distributively to the inner nodes forming the joint.

Flexional forces (fast method)

The method detailed above for estimating the bending moment of the cloth at the joints between sections requires only a modest amount of computation. However, by taking the assumption of symmetry between sections a little further, a greater speed advantage can be achieved. The effect of bending tension between jointed sections can be compared to the effect of a (non-linear) spring connected between the outer nodes of the sections (Fig. 2.12). By considering the geometry of the system, the internal 'strain' (*i.e.* the extent of elongation-compression) of this imaginary spring can be used to compute a value for the bending moment.

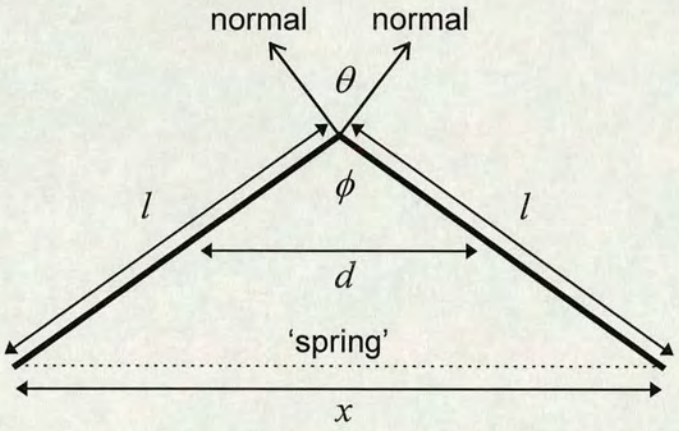



Figure 2.12: Flexional force modelled by non-linear 'spring'

Referring to Fig. 2.12, the following geometrical equalities hold if, as before, $d$ is the distance between the midpoints of the sections:

$$d = \frac{x}{2} = l\sin\frac{\phi}{2} = l\cos\frac{\theta}{2} \tag{2.12}$$

From (2.10) and (2.12), the estimated radius of curvature therefore equates to:

$$R = \frac{d}{2\sin\frac{\theta}{2}} = \frac{l\cos\frac{\theta}{2}}{2\sin\frac{\theta}{2}} = \frac{l}{2\tan\frac{\theta}{2}} \tag{2.13}$$

Thus the bending moment is computed from (2.11) to be:

$$M = K\frac{2}{l}\tan\frac{\theta}{2} \quad \text{where} \quad K = D\cdot E\cdot\frac{T^3}{12(1-\nu^2)} \tag{2.14}$$

Considering, now, the 'strain' $\varepsilon_x$ within the imaginary spring:

$$\varepsilon_x = \frac{x}{2l} - 1 = \cos\frac{\theta}{2} - 1 \tag{2.15a}$$

$$\cos\frac{\theta}{2} = \varepsilon_x + 1 \tag{2.15b}$$

$$\sin\frac{\theta}{2} = \sqrt{1-(\varepsilon_x+1)^2} = \sqrt{-\varepsilon_x(\varepsilon_x+2)} \tag{2.16}$$

Hence, from (2.16) and (2.15b):

$$\tan\frac{\theta}{2} = \frac{\sqrt{-\varepsilon_x(\varepsilon_x + 2)}}{\varepsilon_x + 1} \tag{2.17}$$

and from (2.14) and (2.17):

$$M = K\frac{2}{l}\left(\frac{\sqrt{-\varepsilon_x(\varepsilon_x + 2)}}{\varepsilon_x + 1}\right) \tag{2.18}$$

The magnitude and direction of the resultant forces on the outer nodes of the sections are computed in the same way as for the original method. It may seem at first sight that this second method holds little advantage in terms of required computation. The actual amount of arithmetical operations required by each method is detailed in Table 2.1. There appears to be little difference in this respect. In practice, however, there is a noticeable difference between the methods when implemented on different platforms. Full details of these results are given later in the chapter.

| method | add | subtract | multiply | divide | square root |
|---|---|---|---|---|---|
| *standard* | 6 | 7 | 18 | 3 | 2 |
| *fast* | 7 | 7 | 15 | 5 | 2 |

Table 2.1: Comparison of arithmetic operations required for bending force calculations

Frictional forces

The frictional effects resulting from contact between the cloth and the mannequin body are computed according to a single parameter $\mu$, the coefficient of friction between the two objects. The calculations used within the FIGMENT scheme are detailed in the following chapter (Section 3.8) as part of the discussion of the dynamic response to collisions.

Air resistance

The FIGMENT scheme implements a simple model for the effect of air resistance; a force is applied to each individual node proportionally to the squared magnitude of its velocity and in directional opposition.

$$F_{air} = -k_{air} \cdot |\bar{v}| \cdot \bar{v} \tag{2.19}$$

Wind effects

The effect of moving air on draped fabric is an advantageous feature in that it gives the observer additional insight into the nature of the material which he or she might not have gained from the purely static view of the cloth with its folds, creases, stretching and texture. For this reason, a simple wind simulation feature is included in the FIGMENT scheme. The wind model comprises of a 'pressure' acting in a particular direction on the sections of the cloth meshes in a sinusoidal fashion. In order to simulate the changeability of natural wind or breezes, the phase of the sinusoid is altered discontinuously at random moments (Fig. 2.13). Three parameters are therefore required to specify the wind effect for a particular simulation: a vector determining the direction and magnitude of the wind pressure, the period of oscillation of the effect, and a value indicating the probability that the wind will 'change' during any one iteration of the simulation. (Clearly, the choice of value for the final parameter must take account of the time-step being used for the iteration.)

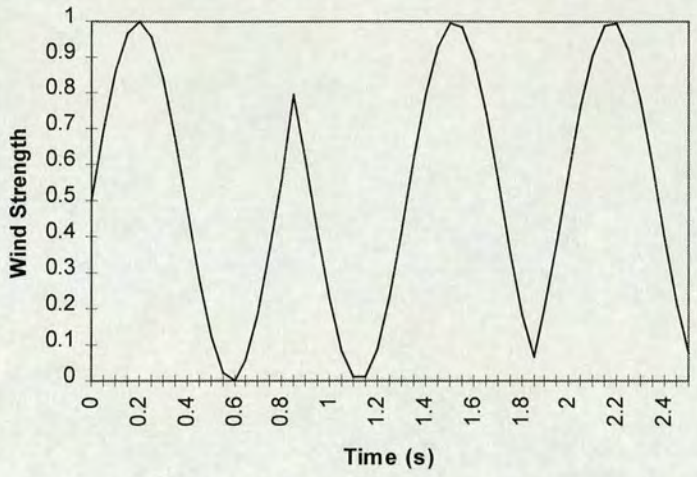



Figure 2.13: Variation in magnitude of wind effect

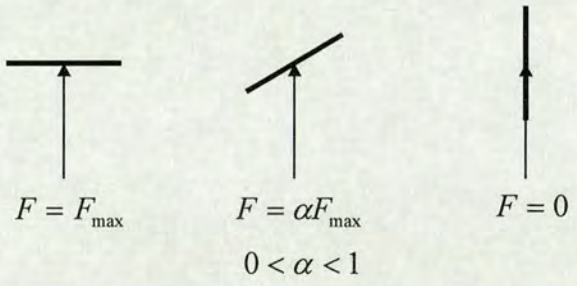



Figure 2.14: Attenuation of force according to orientation of section *(plan view)*

In order to more accurately reflect the results of the wind acting on the mesh sections, the force of the wind is attenuated proportionally to the extent to which the section is facing towards the direction of the wind (Fig. 2.14). The attenuation factor is obtained by taking the absolute value of the dot product of the section normal vector and the direction vector for the wind. The resultant force is applied perpendicularly to the section according to its area, and is computed thus:

$$F_{wind} = \rho_{wind} \cdot \left(\overline{u}_{wind} \cdot \overline{n}_{sect}\right) \cdot A_{sect} \cdot \tfrac{1}{2}\left(1 + \sin\frac{2\pi t}{T_{wind}}\right) \qquad (2.20)$$

where $\rho_{wind}$, $\overline{u}_{wind}$, $T_{wind}$, $\overline{n}_{sect}$ and $A_{sect}$ are respectively the 'pressure', direction, and period of the effect, and the normal vector and area of the section.

When using a wind effect during simulation, it is preferable to allow for a small time delay before applying the effect; this allows for the initial gravitational drop and

collision correction of the cloth to be unhindered by the additional (usually horizontal) forces.

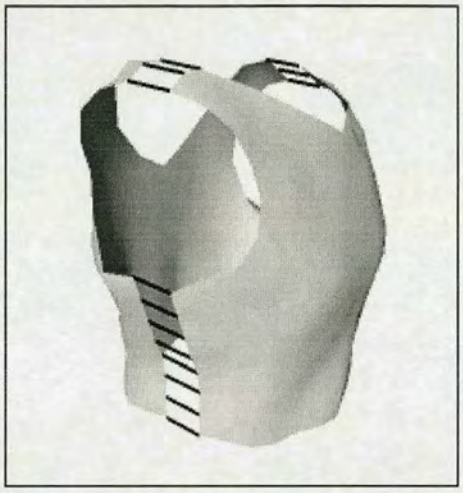


Figure 2.15: Seaming forces between meshes of clothing model

## 2.4 Seaming and hanging forces

In addition to the six basic physical forces detailed above, two further forces are occasionally required for the modelling of clothing items. The first is required for those clothing models which consist of multiple meshes held together with 'virtual seams', directly analogous to the majority of real-life items which are manufactured as discrete patches of fabric and sewn into shape.[2] During physical simulation, the 'seamed' edges of the meshes are held together by 'seaming' forces which act between the corresponding pairs of nodes of the meshes (Fig. 2.15). In the FIGMENT model, the attractive force applied between any pair of nodes is linearly proportional to the distance $d$ between them, up to a specified maximum value:

$$F_{seam}(d) = \begin{cases} F_{\max} \frac{d}{d_{\max}} & d < d_{\max} \\ F_{\max} & d \geq d_{\max} \end{cases} \qquad (2.21)$$

[2] Virtual clothing items used by modelling applications are usually obtained by means of 'virtual tailoring' software, which allows the design and construction of garments in an analogous fashion to that of real-life manufacture. The FIGMENT scheme was developed and tested using models created with the **Fashion Studio V4.0** software suite, produced by Dynamic Graphics Systems.

The second additional force—the 'hanging' force—is required for those clothing items which need to be 'pinned' along certain edges of their meshes, *e.g.* the belt-line of a pair of trousers. The hanging forces act in much the same way as the seaming forces, according to the distance between hanging nodes and a specified vertical height. This vertical height may either be specified as an absolute height in global space or, more usefully, relative to a particular part of the mannequin body, *e.g.* the waist or hips.

## 2.5 Speed comparisons

In order to gauge the speed increases afforded by using the faster methods for calculating tensional and bending forces, the following typical modelling simulation was run on three different platforms—a 200MHz Pentium PC, a 170MHz Sun ULTRASparc and a 180MHz MIPS R5000 Silicon Graphics O2—using each combination of 'standard' and 'fast' force computation methods. The simulation scene was that of a male mannequin being clothed with a 1700-polygon sweater and a 1300-polygon pair of trousers; the time-step for the iterations was 0.001 'virtual' seconds. Tables 2.2, 2.3 and 2.4 indicate the average real time required for the computation of the *internal* (tensional and flexional) forces during each iteration, the average percentage speed increase, and the average percentage of the *total* computation[3] devoted to the computation of those internal forces.

It will be observed that the faster methods prove more advantageous on one platform than another. These differences must be ascribed to the characteristics of the various math processors employed in relation to the types of arithmetical calculations being performed. This contributing factor, as well as those of speed and accuracy requirements, should therefore be taken into account when opting for a particular combination of force calculation methods.

[3] That is, the total computation devoted to simulating the physical dynamics of the cloth meshes (and thus not including rendering time). In order to provide meaningful results in the context of the overall FIGMENT scheme, the 'radial depth' collision method (see Chapter 4) was used in the simulations.

| Methods | Average computation time per iteration | Average speed increase | Average proportion of total computation |
|---|---|---|---|
| standard tensional standard flexional | 53.8 ms | — | 59.1 % |
| fast tensional standard flexional | 43.3 ms | 24.2 % | 53.6 % |
| standard tensional fast flexional | 37.5 ms | 43.5 % | 50.1 % |
| fast tensional fast flexional | 27.8 ms | 93.5 % | 42.6 % |

Table 2.2: Average speed increase when using faster methods (Pentium PC)

| Methods | Average computation time per iteration | Average speed increase | Average proportion of total computation |
|---|---|---|---|
| standard tensional standard flexional | 38.0 ms | — | 27.4 % |
| fast tensional standard flexional | 30.5 ms | 24.6 % | 23.2 % |
| standard tensional fast flexional | 29.6 ms | 28.4 % | 22.7 % |
| fast tensional fast flexional | 22.2 ms | 71.2 % | 18.0 % |

Table 2.3: Average speed increase when using faster methods (Sun ULTRASparc)

| Methods | Average computation time per iteration | Average speed increase | Average proportion of total computation |
|---|---|---|---|
| standard tensional standard flexional | 80.3 ms | — | 48.3 % |
| fast tensional standard flexional | 59.2 ms | 35.6 % | 40.8 % |
| standard tensional fast flexional | 60.2 ms | 33.4 % | 40.8 % |
| fast tensional fast flexional | 39.5 ms | 103.3 % | 31.4 % |

Table 2.4: Average speed increase when using faster methods (Silicon Graphics O2)

The extent of the relative inaccuracies introduced by the faster methods will be discussed later in the chapter (Section 2.7).

## 2.6 Dealing with instability

A highly accurate simulation using the iterative methods detailed above would require a small value of time-step and double-precision floating point operations; these factors, however, would result in a lengthy real-time duration for the simulation. The aim of the FIGMENT scheme is to reduce simulation times and therefore larger values of time-step and single-precision floating point operations are used in practice. As is the nature of iterative methods, this introduces errors into the calculations which, in extreme cases, can result in fatal instabilities—what might be described as 'the exploding clothing problem'. FIGMENT therefore implements two instability-countering measures at different levels within the computation.

Firstly, FIGMENT limits the magnitude of tensional strain existing within the sections of the cloth meshes. Larger iterative time-steps mean that, at times, excessively high tensional forces can be calculated to be acting on individual cloth sections which are thus treated as if acting at such magnitude throughout the period of time-step. In reality, these forces would only act instantaneously and subsequently reduce as the cloth adjusted towards its equilibrium state. In the simulation, however, these forces result in unstable oscillations which prove terminal to the modelling process. FIGMENT therefore limits, to a specified range, the values of normal and shear strain ($\varepsilon_x$, $\varepsilon_y$ and $\tau_{xy}$) calculated from the current state of each cloth section. In this way, the resultant internal forces are prevented from reaching unrealistic magnitudes.

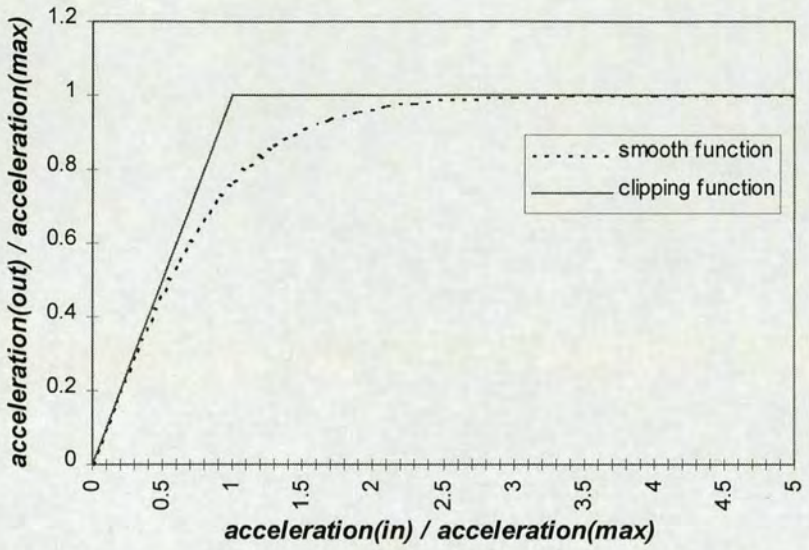



Figure 2.16: Acceleration limiting functions

Secondly, FIGMENT also limits the magnitude of the resultant acceleration of each node within the cloth meshes, as computed at the end of each iteration. After calculating the resultant force on each node according to all the aforementioned internal and external factors, the resultant acceleration is capped at a specified maximum value by means of a limiting function. The function originally implemented by the FIGMENT scheme used a hyperbolic tangent function to smoothly limit the acceleration as it reached its maximum value (Fig. 2.16, 'smooth function'). However, it was found that a simpler function which merely 'clips' values exceeding the specified maximum (Fig. 2.16, 'clipping function') gave results in practice which differed negligibly from those obtained using the more sophisticated function (see Section 2.8). FIGMENT therefore uses the latter, less computationally intensive, function for the purposes of limiting acceleration values.

A question must be asked, then, as to the cost of using these instability-countering measures. What is the extent of the errors introduced into the simulation results? These issues are answered, by comparing the results of example simulations, in the following sections.

## 2.7 Measuring accuracy

At this point, it would be informative to objectively assess the relative accuracy of simulations which use faster (but less precise) methods of force computation, or those which employ instability-countering measures. In order to do so, a technique is required that indicates the deviation, in some respect, of a test case from a control case over the relevant period of simulation. The following approach has been used during the development of the FIGMENT scheme.

While running the simulation under inspection, the state of the cloth meshes (*i.e.* the positions of the mesh nodes) within the modelling scene is recorded at regular intervals throughout. An appropriate control simulation is also run in which the mesh states are recorded in the same fashion. The two series of meshes states are then compared by computing *(a)* the mean distance between corresponding nodes in each pair of meshes, and *(b)* the standard deviation of the distance between corresponding nodes. These two values are plotted on one graph with respect to simulation time on the $x$-axis; the first value indicates the general extent of error as the simulation progresses, the second indicates the simultaneous extent of 'spread' of the error. The actual values plotted are more meaningful when considered in relation to some standard dimension of the simulation; in the examples given below (and throughout this thesis), the values are expressed as a percentage of the height of the overall mannequin on which the clothing items are modelled. Thus, for a mannequin representing a 5'10" person, a 1% error corresponds to a node deviation approximately equivalent to 1.8cm.

As a further insight into the distribution of error at any stage of the simulation, the distances between corresponding nodes can be plotted on a three-dimensional graph which indicates the *frequency* of nodes such that the distance error lies between certain ranges. In the error distribution graphs detailed in following sections, the *maximum* error which occurred through out the whole simulation has been determined; one hundred equally-spaced error ranges between zero and this maximum value have then been used to compute the frequencies of nodes falling into each range and to plot those values against the vertical axis.

These two types of accuracy analysis can therefore provide an objective basis for determining the inaccuracies introduced by specific speed-gaining variations in the simulation parameters. However, bearing in mind the purposes of the FIGMENT scheme in producing useful virtual representations of modelled clothing, a *subjective* comparison of the visual results of a test simulation and its control is also a valuable and highly relevant method of assessing accuracy. For this reason, the final rendered frames of the majority of example simulations are provided throughout this thesis.

## 2.8 Results

The results of test simulations using five different FIGMENT modelling configurations are detailed in this section. The first two are intended to assess the error incurred in simulations performed with a larger time-step and instability-countering measures when compared with a control simulation performed with a small time-step (and thus requiring no instability-countering measures). The first of the two uses the original 'smooth' acceleration limiting function; the second uses the simple 'clipping' function.

The final three simulation configurations are intended to assess the error incurred in performing simulations with either, or both, of the faster force calculation methods detailed in Section 2.3. The first of the three configurations uses the faster method for tensional force calculation; the second uses the faster method for bending force calculation; the third uses both methods together. In each case, the comparison is made with a control configuration which employs neither of the faster methods.

Accuracy results are provided here for two typical modelling scenes, each simulated using six modelling configurations: the five described above plus one control simulation. The first scene is identical to that used for the speed comparisons of Section 2.5; a male mannequin being clothed with a 1700-polygon sweater and a 1300-polygon pair of trousers. The second scene consists of a female mannequin being clothed with a 1100-polygon dress and an 840-polygon jacket. Table 2.5 indicates the full details for each simulation performed in the following results. In

every case, the simulation was performed for 6.0 'virtual' seconds. All results were obtained on a 170MHz Sun ULTRASparc platform.

| Simulation | Description | Time-step | Instability-countering measures | Force computation methods |
|---|---|---|---|---|
| *1A* | Male with sweater and trousers | 0.1 ms | (unnecessary) | standard tensional<br>standard flexional |
| *1B* | Male with sweater and trousers | 1 ms | strain limiting +<br>'smooth' accel. limiting | standard tensional<br>standard flexional |
| *1C* | Male with sweater and trousers | 1 ms | strain limiting +<br>'clamping' accel. limiting | standard tensional<br>standard flexional |
| *1D* | Male with sweater and trousers | 1 ms | strain limiting +<br>'clamping' accel. limiting | faster tensional<br>standard flexional |
| *1E* | Male with sweater and trousers | 1 ms | strain limiting +<br>'clamping' accel. limiting | standard tensional<br>faster flexional |
| *1F* | Male with sweater and trousers | 1 ms | strain limiting +<br>'clamping' accel. limiting | faster tensional<br>faster flexional |
| *2A* | Female with dress and jacket | 0.1 ms | (unnecessary) | standard tensional<br>standard flexional |
| *2B* | Female with dress and jacket | 1 ms | strain limiting +<br>'smooth' accel. limiting | standard tensional<br>standard flexional |
| *2C* | Female with dress and jacket | 1 ms | strain limiting +<br>'clamping' accel. limiting | standard tensional<br>standard flexional |
| *2D* | Female with dress and jacket | 1 ms | strain limiting +<br>'clamping' accel. limiting | faster tensional<br>standard flexional |
| *2E* | Female with dress and jacket | 1 ms | strain limiting +<br>'clamping' accel. limiting | standard tensional<br>faster flexional |
| *2F* | Female with dress and jacket | 1 ms | strain limiting +<br>'clamping' accel. limiting | faster tensional<br>faster flexional |

Table 2.5: Details of example simulations

Fig. 2.17(a) plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes in simulations *1B* and *1C* compared with the more accurate simulation *1A*. Fig. 2.17(b) plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes in simulations *2B* and *2C* compared with the more accurate simulation *2A*. In both cases, the error (*i.e.* distance between corresponding nodes) is expressed as a percentage of the total height of the mannequin. In Fig. 2.17(b) and other subsequent error plots in this thesis, it may be observed that some of the error values continue to rise (or fall) at the end of the simulation period. This is due to the fact that a proportion of the cloth sections in the scene were still in motion right up until this point; if the simulation had been continued, the error values would be seen to 'flatten out'.

Fig. 2.18(a) plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes for simulation *1C* compared with simulation *1B*. Fig. 2.18(b) plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes for simulation *2C* compared with simulation *2B*.

Fig. 2.19 plots the *distribution* of error between corresponding nodes (a) for simulation *1B* compared with simulation *1A* and (b) for simulation *1C* compared with *1A*. As before, the error is expressed as a percentage of the total height of the mannequin.

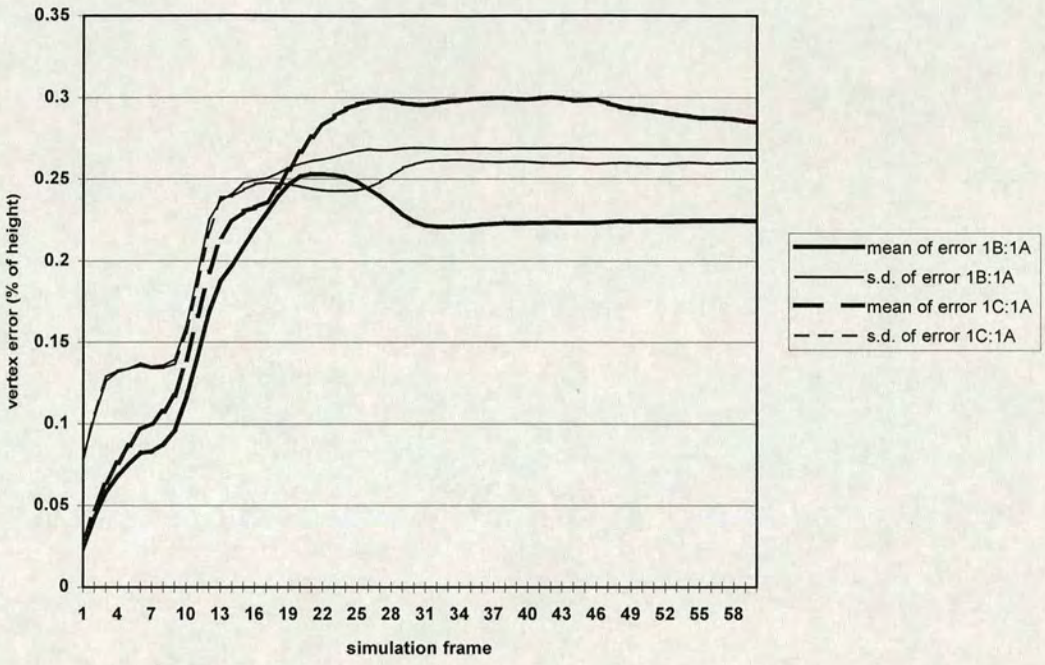



Figure 2.17(a): Accuracy of simulations *1B* and *1C* relative to *1A*

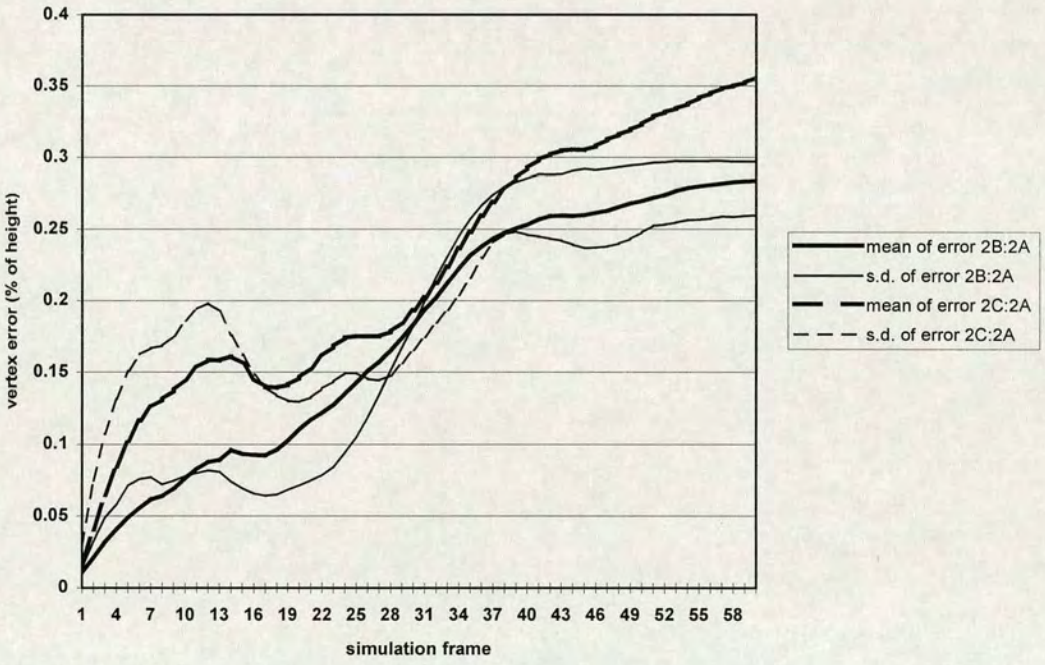



Figure 2.17(b): Accuracy of simulations *2B* and *2C* relative to *2A*

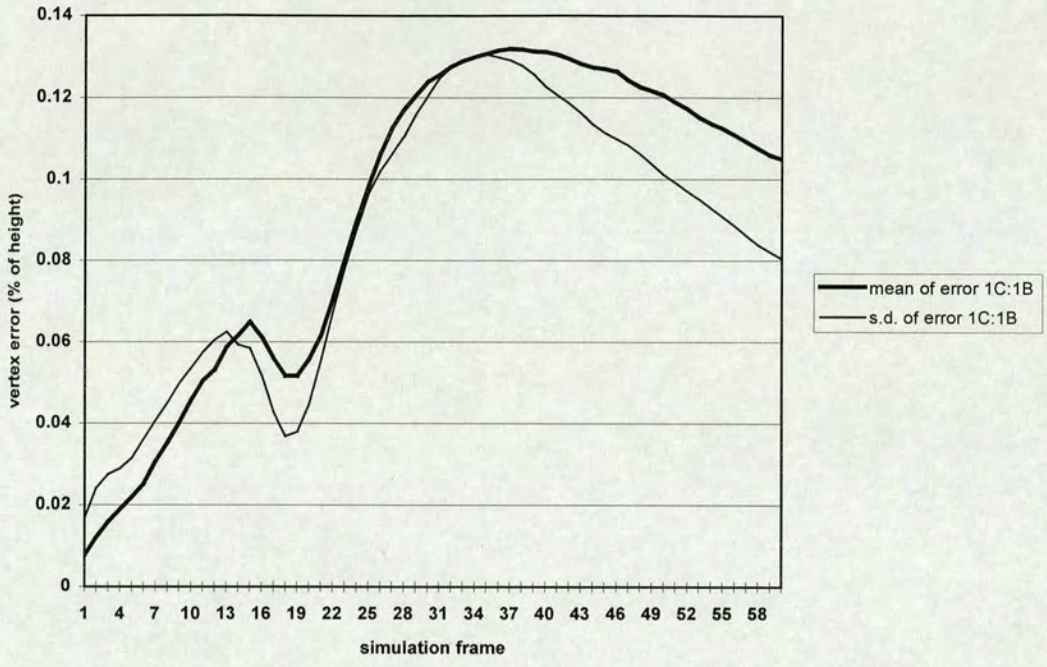



Figure 2.18(a): Accuracy of simulation *1C* relative to *1B*

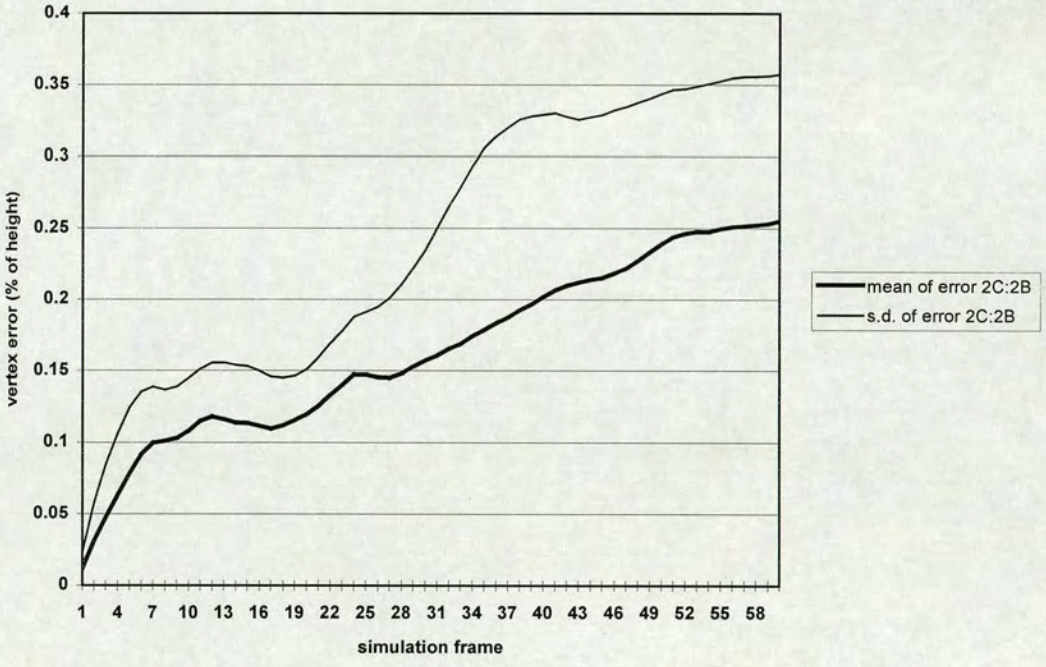



Figure 2.18(b): Accuracy of simulation *2C* relative to *2B*

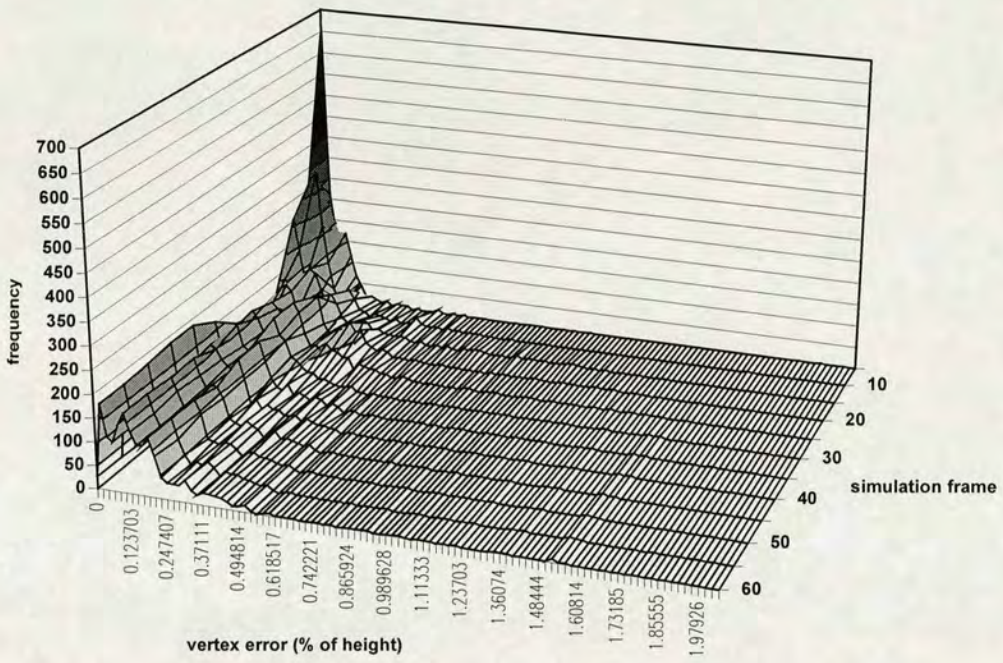



Figure 2.19(a): Distribution of error for simulation *1B* relative to *1A*

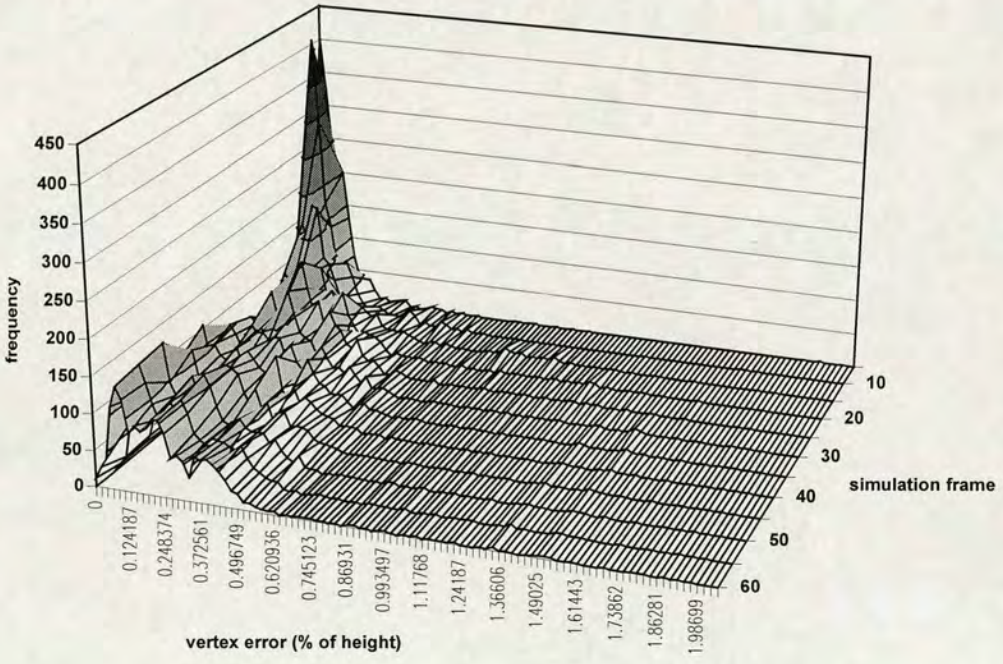



Figure 2.19(b): Distribution of error for simulation *1C* relative to *1A*

To illustrate the cost in accuracy incurred by the faster force computation methods, Fig. 2.20(a) plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes in simulations *1D*, *1E* and *1F* compared with simulation *1C*. Fig. 2.20(b) plots the same for *2D*, *2E* and *2F* compared with *2C*.

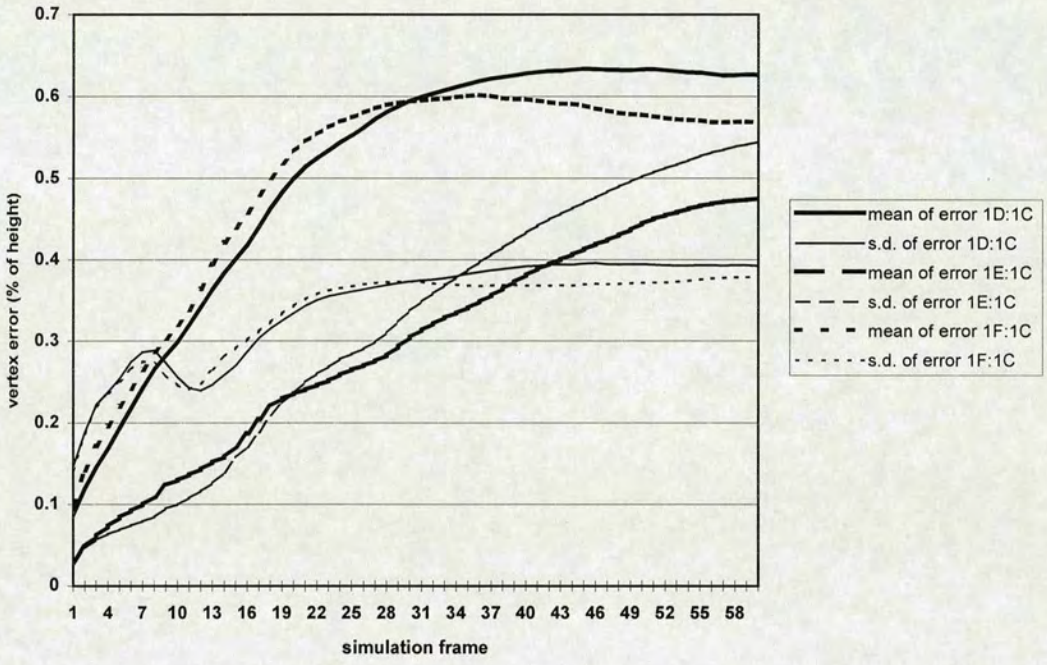



Figure 2.20(a): Accuracy of simulations *1D*, *1E* and *1F* relative to *1C*

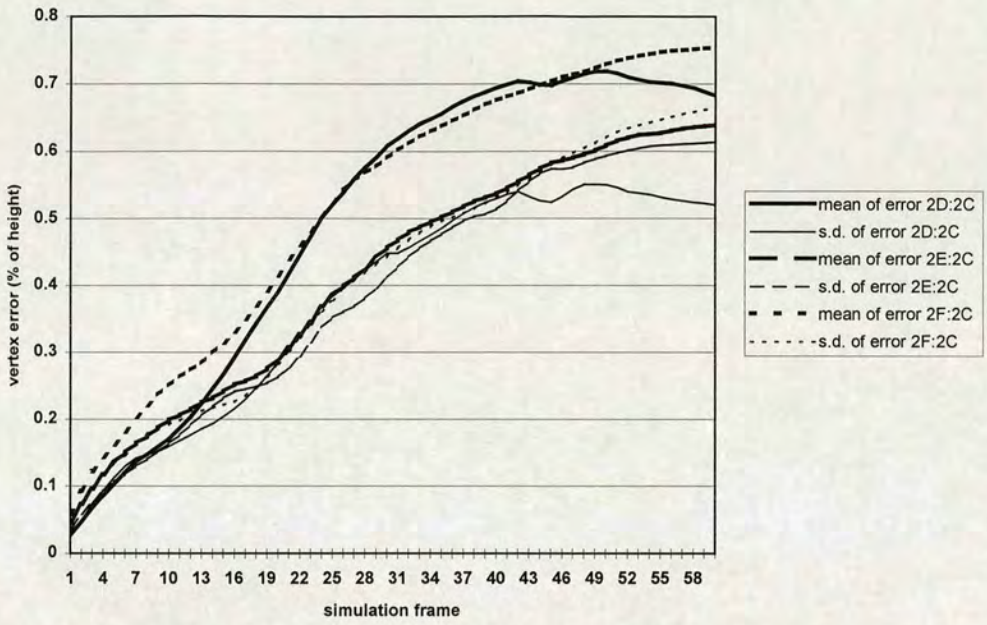



Figure 2.20(b): Accuracy of simulations *2D*, *2E* and *2F* relative to *2C*

Fig. 2.21 plots the *distribution* of error between corresponding nodes (a) for simulation *1D* compared with simulation *1C*, (b) for simulation *1E* compared with *1C*, and (c) for simulation *1F* compared with *1C*.

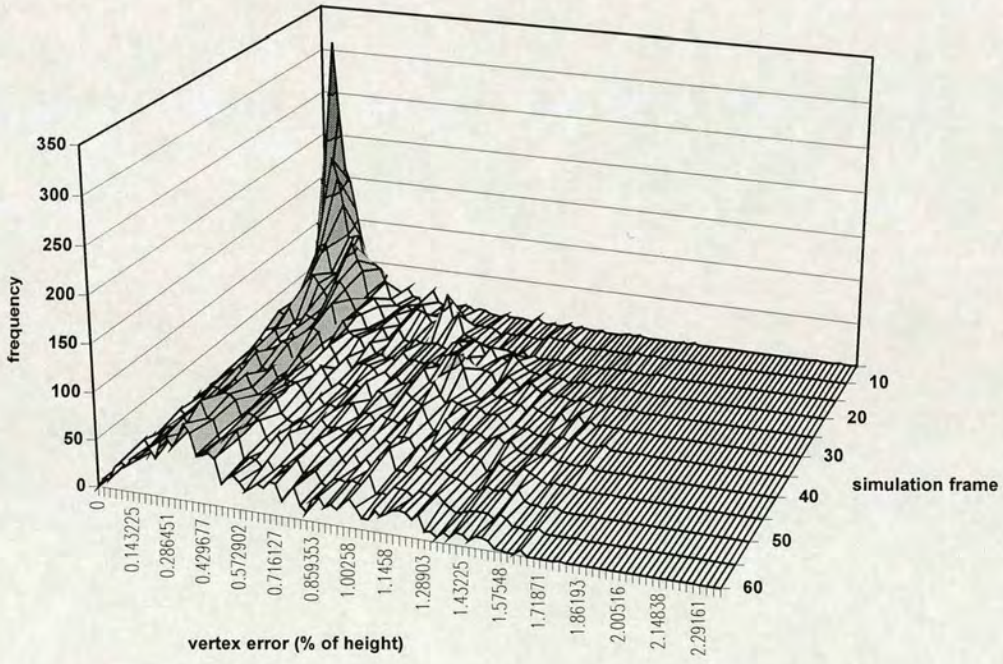



Figure 2.21(a): Distribution of error for simulation *1D* relative to *1C*

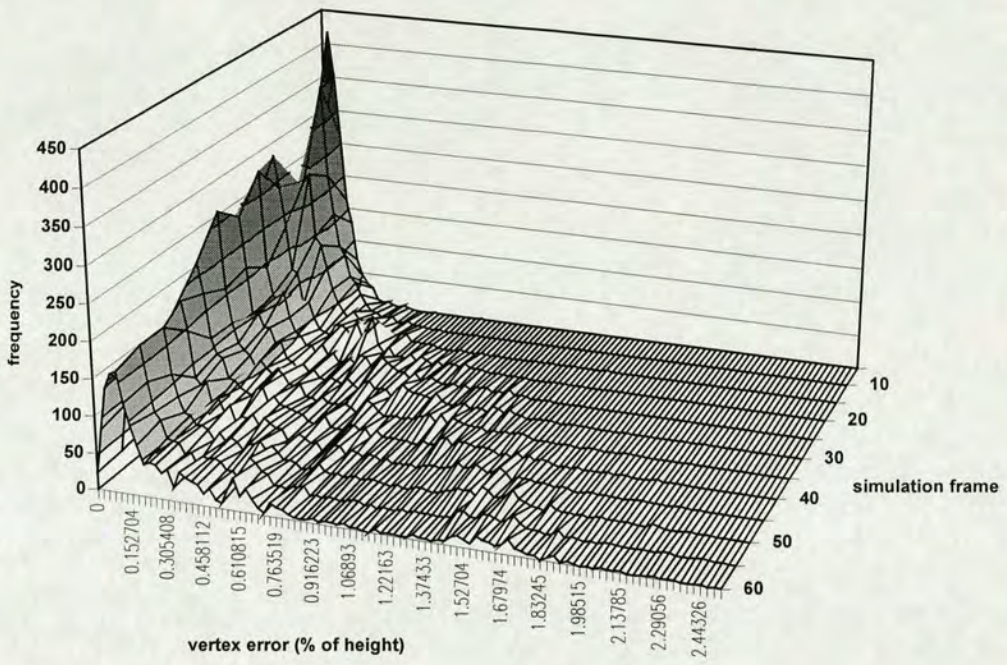



Figure 2.21(b): Distribution of error for simulation *1E* relative to *1C*

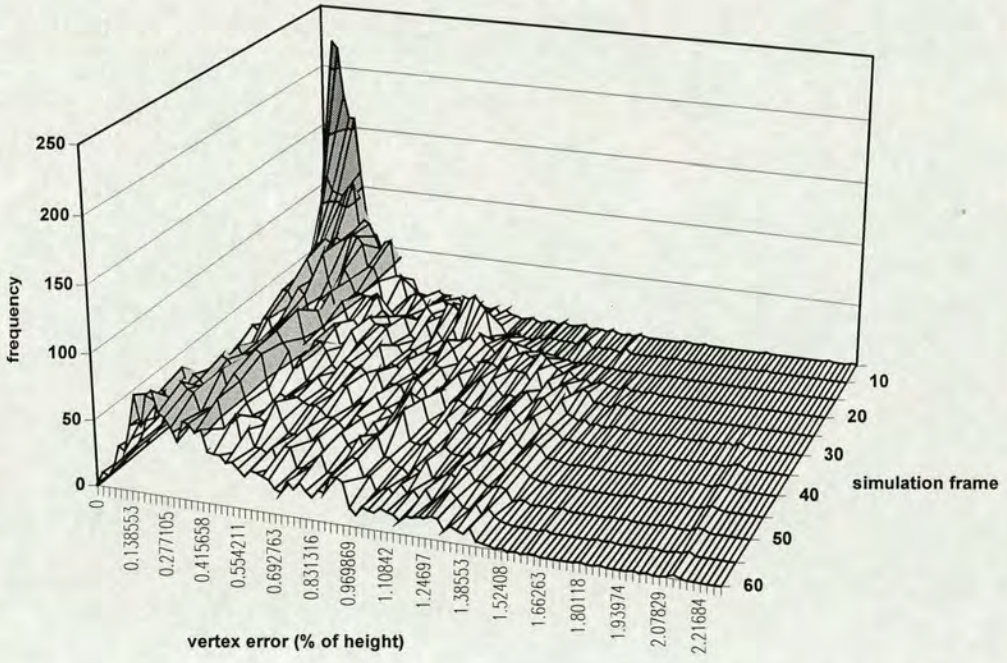



Figure 2.21(c): Distribution of error for simulation *1F* relative to *1C*

Finally, to give an indication of the general absence of visible differences between the results of the example simulations, Fig. 2.22(a) shows the final frames of

Finally, to give an indication of the general absence of visible differences between the results of the example simulations, Fig. 2.22(a) shows the final frames of simulations *1A*, *1C* and *1F*, while Fig. 2.22(b) shows the final frames of simulations *2A*, *2C* and *2F*.

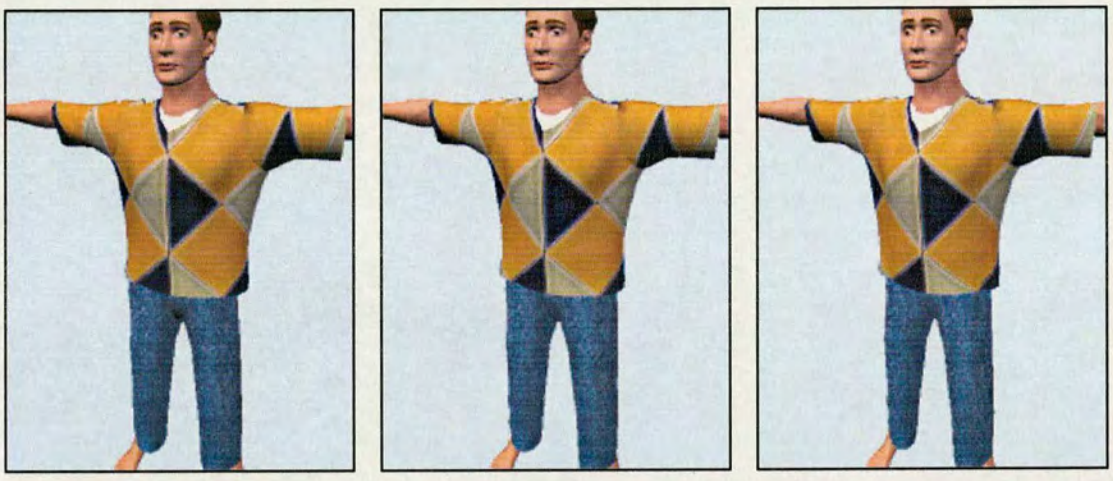


Figure 2.22(a): Final frames of simulations *1A*, *1C* and *1F (left to right)*

Figure 2.22(b): Final frames of simulations *2A*, *2C* and *2F (left to right)*

## 2.9 Conclusions

Although the standard physical model for the FIGMENT scheme provides a fast and robust set of calculations for the dynamic simulation of seamed clothing items, the timing results of Tables 2.2, 2.3 and 2.4 indicate that the use of faster methods for determining the tensional and flexional forces acting on discrete cloth sections can approximately double the speed of computation. Moreover, an analysis of the accuracy of results obtained when using these faster methods indicates that the error introduced is relatively small. The faster methods therefore provide an useful and reliable alternative in applications where modelling speed is a crucial factor. In cases where significant folding of loose cloth occurs (as in the second example modelling scene) the visual results (Fig. 2.22(b)) indicate that the faster flexional force computation allows a slightly greater degree of bending in such garments; further work could therefore be directed toward compensating (in some computationally efficient way) for this effect.

In addition, the implementation of simple instability-countering measures can permit the use of both a larger time-step and single-precision floating point arithmetic which can substantially reduce simulation times. The accuracy analysis and visual results provided in this chapter demonstrate that the error introduced by this strategy is quite insignificant, yet without the instability-countering measures, such simulations would fail altogether.

This first point of the FIGMENT scheme therefore provides an excellent foundation for the implementation of an interactive garment modelling service, by enabling maximal simulation speeds whilst minimizing the inaccuracies introduced. The degree to which simulation speed is traded against fidelity may be varied according to a number of contributing factors—the complexity of the cloth meshes used, the simulation time-step, the choice of force computation methods—in order to obtain optimum results for any particular implementation.

## 2.10 Summary

This chapter has described the physical model for the FIGMENT scheme, comparing it with other approaches to dynamic cloth simulation and establishing the suitability and advantages of using a discrete particle-based iterative model as the basis for the scheme. The necessary calculations for computing the internal and external forces involved in the physical model were provided in detail, including two alternative methods for faster determination of tensional and flexional forces. The use of instability-countering measures within the scheme was discussed, with subsequent results demonstrating that simple 'clipping' functions applied to certain values involved in the compution of cloth mesh dynamics allow accurate simulations with substantially reduced computation times. Accuracy analysis and visual results were provided for two typical modelling scenes obtained from simulations with various combinations of time-step, instability-countering measures and force computation methods.

# Chapter 3

# Collision Approximation: Capsule Method

## 3.1 Introduction

Collision handling techniques must play a crucial role within any system designed to simulate the modelling of virtual clothing. Primarily, the colliding of cloth with the surface of a mannequin must be efficiently detected and an appropriate dynamic response applied to the physical elements of the model. In addition, the collision of the cloth with itself and other items of clothing must be considered in order to avoid visible interpenetration of cloth surfaces and other anomalies.

However, the implementation of collision handling methods can cause substantial difficulties for any modelling system in which a maximal simulation speed is desired since, in the nature of the case, the computational requirement of methods which consider potential collisions on a vertex-to-polygon or polygon-to-polygon basis increases according to a square law with the number of discrete elements in a modelling scene. In practice, the implementation of collision detection and response methods can account for anything up to 90% of the total computation required for a typical simulation.

Although a number of efficient collision detection methods have been developed previously, none of those which meet the special requirements of cloth modelling provide the performance gain necessary for the applications which would implement the FIGMENT scheme. Therefore, in this chapter and the following one, two novel methods for collision handling based on an approach known as 'volume approximation' are presented in detail which allow for a dramatic decrease in

computational requirement by sacrificing some fidelity of surface representation with respect to the mannequin body. These methods provide two levels of compromise between relative speed and accuracy. For any particular application case, a choice between (or a combination of) the methods must be chosen according to the available hardware capabilities and the desired fidelity of results.

This chapter describes the first of the two methods, known as the 'capsule' method of collision approximation, which aims to represent the volume of the mannequin body using a series of relatively simple 'capsule' structures. The development of the capsule structure is described in detail, along with the genetic algorithm used to determine the best-fitting combination of such structures for any particular mannequin body. Finally, the results of using this particular method are assessed by comparison with a standard polygon-based collision handling algorithm, in terms of both relative speed and accuracy.

## 3.2 Collision handling methods

Approaches to collision detection and response can generally be divided into two categories. In the first place, there are those methods directed towards non-real-time applications such as computer generated film animations (*e.g.*, Carignan *et al*, 1992, Volino *et al*, 1996) or mechanical simulations. In these cases, the primary requirement is that of precision rather than speed, although any avenues for optimizing those methods will naturally be welcomed. Secondly, there are those techniques developed for use within real-time interactive applications such as virtual reality environments or robotics. The requirement of these techniques is to reduce computation times at the expense of accuracy (to a minimal extent) in order to maintain acceptable frame rates throughout the simulation. Naturally, the methods of both categories are of significance to the FIGMENT scheme, which aims to supply an acceptable level of accuracy with a response as near to real-time as possible.

Methods for non-real-time applications

Although somewhat dated now, Moore and Wilhelms (1988) provide a useful overview of approaches to both collision detection and response, presenting in detail two particular collision detection algorithms: one for flexible surfaces (such as cloth meshes) and one for convex polyhedra. The former algorithm works on a "points versus triangles" basis, *i.e.* the positions of vertices within the meshes are examined at the beginning and end of a time-step of animation to see if any have passed through a mesh polygon. By restricting mesh polygons to triangles, problems of surface ambiguity are avoided. In order to detect both the penetration of one surface by another and surface self-penetration, every vertex must be tested against every polygon which does not include that vertex. Thus the basic algorithm has $O(nm)$ complexity for $n$ polygons and $m$ vertices. Moore and Wilhelms go on to describe the use of bounding boxes arranged in an octree structure to reduce the time requirement to $O(m \log m)$ to construct the octree and $O(n \log m)$ to traverse it when detecting collisions. For dynamic surfaces, the octree must be reconstructed before each phase of collision detection.

Also falling into the first category is the algorithm for determining collisions between time-dependent parametric surfaces detailed by Von Herzen, Barr and Zatz (1990). By imposing constraints on the rates of change of the surfaces (*i.e.* ensuring a maximum velocity for any point on a surface), it is possible to compute a set of bounding volumes for each surface between two points in time and thus to quickly eliminate non-colliding pairs of surfaces. An adaptive surface sampling algorithm is then employed to narrow down points of collision to within a specified accuracy $\gamma$, *i.e.* pairs of surfaces which come within a distance $\gamma$ of each other at some point will be detected as colliding. The algorithm proves to be of particular use in computer animations where deformable dynamic characters or objects are represented as sets of such parametric surfaces, *e.g.* Chadwick *et al* (1989).

Another non-real-time, but highly efficient, approach to collision handling is that offered by Yang and Magnenat Thalmann (1993). The algorithm was developed specifically for the purposes of modelling clothing using virtual actors and uses a

method based on octree-subdivision of polygon sets to narrow down the field of collision cases. Similar to that of Moore and Wilhelms (1988), the algorithm can handle both penetrations between discrete surfaces and surface self-penetration, although penetrations are detected on a polygon-to-polygon basis rather than a vertex-to-polygon basis. For a scene containing $n$ potentially colliding polygons, the algorithm takes $O(n \log n)$ time to build the octree structure and $O(n \log n)$ time to search it for collisions. Also originating from MIRALab at the University of Geneva, Volino and Magnenat Thalmann (1994) present an algorithm for the efficient detection of cloth self-collision, based on geometrical shape regularity properties. The paper details a method for building a hierarchical subdivision structure for a highly discretized deformable surface (*i.e.* a polygon mesh) which permits efficient elimination of non-penetrating sets of subsurfaces. In the example scenarios provided, the algorithm detected collisions in $O(n)$ average time for $n$-polygon surfaces.

Methods for real-time applications

Since collision handling for non-trivial virtual environments can be a complex and time-consuming process, a considerable amount of recent work has been directed towards various attempts to either optimize or approximate the calculations involved in detecting and responding in real-time to interactions between polygonal objects within the environment. In practice, with current hardware limitations, approximation is always required and an acceptable level of inaccuracy may be incurred.

Efficient algorithms for contact determination and interference detection between polygonal objects in large-scale interactive virtual environments are presented by Lin (1993), Lin and Manocha (1995), Cohen, Lin, Manocha and Ponamgi (1995) and Ponamgi, Manocha and Lin (1995). The algorithms rely on hierarchical representations of the scene to quickly identify pairs of interacting objects and capitalize on temporal and spatial coherence between successive frames of simulation. Although restricted to interactions between solid (*i.e.* non-deformable)

EX BIBL. UNIV. EDINBURGH

objects, the algorithms permit highly efficient collision detection between large numbers (*e.g.* thousands) of arbitrary polyhedra.

Another hierarchical algorithm for efficient and exact interference detection amongst complex models undergoing rigid motion is introduced by Gottschalk, Lin and Manocha (1996). Arbitrary polyhedra are represented using so-called 'OBBTree' structures, optimized hierarchies of oriented bounding boxes (OBBs). Although OBBs have been widely utilised in the past, this paper offers new algorithms for determining tight-fitting OBBs and an efficient method for checking overlap between a pair of OBBs. The approach is most advantageous for performing collision detection at interactive rates in virtual environments with multiple objects comprised of high numbers (hundreds of thousands) of polygons.

Developed for a similar range of applications, Hubbard (1993, 1995a, 1995b, 1996) presents a method of approximating polyhedra with spheres for real-time collision detection. The technique involves specifying representations of virtual objects using hierarchies of variously-sized spheres which approximate the objects at multiple levels of detail. The root level is simply the bounding sphere of an object; subsequent levels are unions of successively more spheres, approximating the objects at higher resolutions. Using this multi-level approximation structure, the time-critical collision detection algorithm employs a progressive refinement method whereby the refinement only proceeds until the available processing time in one visual frame is exhausted. The detection algorithm performs favourably when compared with the BSP-tree algorithm of Thibault and Naylor (1987), improving performance by factors of 10 to 100.

## 3.3 Assessment of collision handling methods

The requirements of an effective collision handling algorithm for an interactive virtual mannequin lie somewhere between those satisfied by the two categories of algorithms described above. On the one hand, the algorithm must be able to handle arbitrary non-rigid surfaces like those for non-real-time simulation. On the other hand, the algorithm must also make some form of approximation in order to reduce

the simulation times involved and thus to maintain the appropriate level of interactivity and usability. Since the specific application involves collisions between two different types of polygonal objects—highly deformable polygon meshes and rigid convex polyhedra—an approach involving techniques from both categories would seem to be demanded.

The second algorithm given by Moore and Wilhelms (1988), although appropriate for long-term modelling of fabric, is unsuitable on account of the computation required and also the critical role of the simulation time-step for collision detection. Since the FIGMENT scheme aims to allow larger time-steps, the accuracy of collision detection by this method would be considerable reduced.

The method of Von Herzen *et al* (1990) could only be used if both clothing models and mannequin models were to be represented by parametric surfaces rather than polygonal meshes and thus is unsuitable for this application. Although the dynamic modelling of fabric by parametric surfaces is possible in theory, the computations involved would be quite prohibitive, not least for the present application.

The primary disadvantage of the algorithm developed by Yang and Magnenat Thalmann (1993) is the computation required, despite the commendable efficiency of the algorithm in comparision to other polygon-to-polygon detection methods. If cloth self-collision handling were absolutely necessary, the algorithm of Volino and Magnenat Thalmann (1994) would be an excellent choice. However, the value of such a feature in a FIGMENT-based application when weighed against the computational cost involved is debatable, particularly when the advantages of the hybrid rendering algorithm (detailed in Chapter 6) are considered. In practice, it is generally true that the natural rigidity of the simulated fabric prevents self-penetration of any considerable degree. Furthermore, any slight penetrations—either within a cloth mesh or between two meshes—which would otherwise appear as rendering anomalies are masked by the effect of the hybrid rendering algorithm. Thus, the omission of cloth self-collision detection by the FIGMENT scheme is perfectly warranted in view of its aims and practical function. Any cases which might

require such a feature may be catered for by extending the scheme, but must be considered exceptional.

Turning to consider the time-critical collision methods, a different set of difficulties arise. The algorithms of Lin, Manocha, Cohen and Ponamgi, despite being highly efficient are unsuitable for the present application. The reductions achieved by consideration of temporal and spatial coherence, although significant for complex scenes in which only a proportion of objects are in collision at any one time, would not apply to scenarios in which pairs of objects are in continuous contact. Moreover, the algorithms are presently only appropriate for rigid models.

For similar reasons, the speed advantages afforded by the OBBTree algorithm of Gottschalk, Lin and Manocha (1996) would tend not to apply in the case of cloth modelling. Furthermore, the need to rebuild the OBBTree structures for the deformable cloth meshes before each phase of collision detection would be counterproductive. The possibility of a modified algorithm which uses OBBTrees for only the mannequin body polygons would fare little better than the non-real-time algorithms previously mentioned, taking $O(n \log m)$ average time for an $m$-polygon mannequin and $n$-polygon cloth meshes.

The same difficulties would also apply to an implementation of Hubbard's progressive sphere approximation algorithm. The time-critical nature of the algorithm is irrelevant for this application; for accurate modelling the progressive refinement would always be required to proceed to the highest level (*i.e.* the polygonal surface itself) and thus there would be no significant speed advantage.

Having assessed the various approaches to real-time and non-real-time collision handling, a new approach is required for the present application which avoids the problems involved in implementing any particular one of those reviewed above, even with substantial modification. It seems inevitable that approximation must be made in order to avoid the computational cost of polygon-to-polygon or vertex-to-polygon comparisons (even when optimized by hierarchization). However, the use of *multilevel* approximations with *time-critical* refinement (beyond a simple object bounding box check) is inappropriate for an application which does not require a strictly real-time response. An ideal solution would be to use a method of

approximation in which a single level of approximation both afforded a significant reduction in computation requirements whilst providing sufficient accuracy at the surface level to allow a faithful collision response. The technique of collision volume approximation is one such solution.

## 3.4 Collision volume approximation

The basis of a collision volume approximation method is that of representing the three-dimensional shape of a solid object (or, as in this case, the discrete parts of a solid object) by a generic geometrical structure. Hence, an approximation is made to the 'collision volume' of the object which is used in place of the actual object when handling collisions. The design of the structure should allow for a suitable level of correspondence between its surface and that of the original object, as well as providing a relatively simple computational procedure for both *(1)* determining whether a point has penetrated the volume of that structure and also *(2)* calculating relevant information (point of penetration, surface normal vector, *etc.*) from which a dynamic collision response can be established.

To the best knowledge of the author, this particular approach to collision handling has not been previously developed or implemented. There is a superficial similarity to the approach of Hubbard (1993, 1995a, 1995b, 1996) which uses a series of approximations consisting of progressively smaller spheres; however, at the lowest level, the latter method still considers collisions with respect to the original polygonal surface of the object. The collision volume approximation method, in contrast, uses only one level of approximation with no progressive refinement. It would certainly be theoretically possible to develop the FIGMENT method to use a precomputed hierarchy of increasing resolutions of approximation; this would nonetheless appear to defeat the primary advantage presented by using an algorithm which takes $O(n)$ time for collision detection, as do both of the volume approximation methods presented herein.

## 3.5 The 'capsule' structure

The 'capsule' method of collision volume approximation was developed from the observation that there is a general geometric similarity between the individual body parts of a hierarchically jointed mannequin model. Each body part object can be approximated, to a lesser or greater extent, by a cylindrical shape of variable dimensions with more or less rounded ends (Fig. 3.1). The forearm, for example, may be represented by a longer shape with less rounded ends; the neck by a shorter, fatter shape; and the hips and waist by a wider, more rounded shape. It should be clear that the calculations required to detect the penetration by a vertex of such a geometrically simple structure are considerable less than for any arbitrary polyhedron. Similarly, the calculations needed to determine a response to collisions are also straightforward. As indicated previously, there is a superficial similarity in approach to that of Hubbard, who uses spheres for approximation purposes, although the differences are quite significant. In this case, only one structure is tested for penetration, *i.e.* no progressive refinement is performed, and the collision response is computed with respect to the approximation structure itself, rather than the actual polygons of the rigid model.

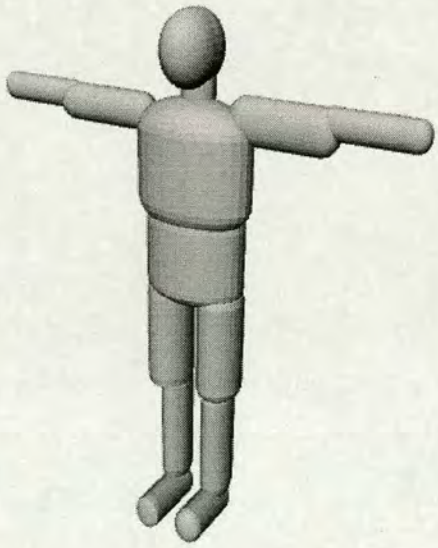


Figure 3.1: Mannequin body approximated using simple geometrical shapes

The collision approximation structure introduced above is referred to as a 'capsule' within the FIGMENT scheme (for obvious reasons), although it has developed to a more complex form since its initial conception in order to produce a more accurate representation of the body parts of the mannequin.

It is informative to relate the development of the capsule structure from its simplistic origins to its more complex specification. The initially conceived capsule was defined by only one parameter $r$, corresponding to the degree of rounding at the ends of the object, where $r = 1$ describes a unit sphere and $r = 0$ describes a unit cylinder (Fig. 3.2) aligned to the vertical axis. The rounded sections consist of hemispheres vertically scaled according to the value of $r$.

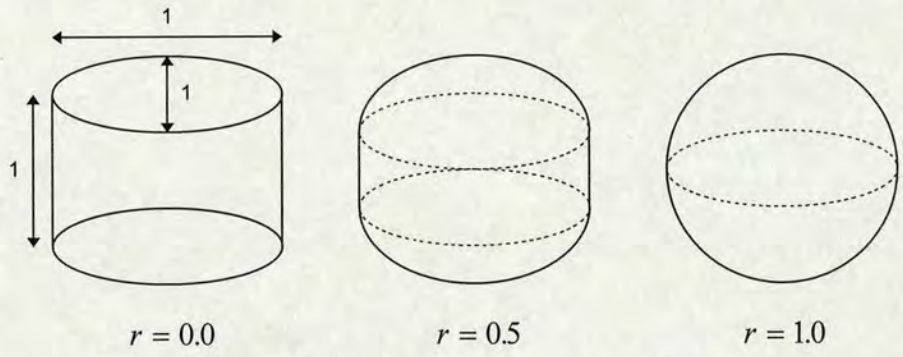



Figure 3.2: Original unit capsule structure

In order to determine the best-fitting capsule for each body part object of the mannequin, the axis-aligned bounding box of that object was first computed, and hence the combined three-dimensional transformation (translation, rotation and scale factors) required to transform the unit capsule into the object's local space. A best-fit algorithm then determined the optimum value for the parameter $r$ in order to minimize the error between the surface of the body part and the surface of the capsule; with only one variable in the best-fitting process, a binary search algorithm was adequate for the task. In this way, the optimum value for $r$ plus the geometric transform obtained from the bounding box were sufficient to specify the capsule object which most closely approximated the volume of the body part object in question.

Evidently, a *vertically* aligned capsule object with whatever optimum values are computed might prove to be a less accurate approximation than an object aligned along one or other of the horizontal axes. For this reason, all three initial orientations of the capsule were considered (by adjusting the rotation and scale components of the geometric transform) during the best-fitting process and the optimum result taken overall.

As would be expected, the initial parameter $r$ on its own did not allow a satisfactorily close approximation of the various body parts, and so the capsule structure was progressively developed by specifying additional parameters to further modify the basic capsule shape and thus provide a better representation of the mannequin body surface. However, as the number of parameters increased, the method of determining optimum values for each body part became more difficult, hence the adoption of a genetic-based best-fitting algorithm as described later in this chapter.

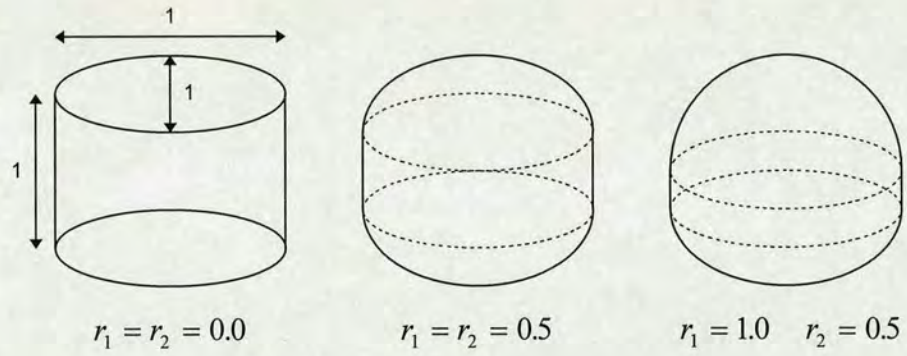
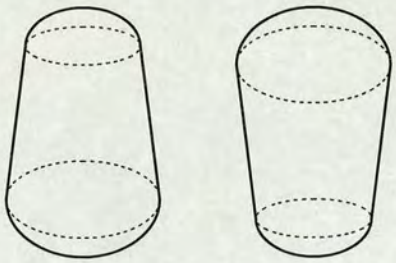



Figure 3.3: Variation of parameters $r_1$ and $r_2$ in unit capsule

The first development involved separating the rounding factors for the ends of the capsule into two values, $r_1$ and $r_2$ (Fig. 3.3). Following this, a tapering parameter $t$ was added, allowing the capsule to become linearly narrower at one end and broader at the other (Fig. 3.4). Thirdly, parameters $x$, $y$, and $z$ were specified to permit the dimensions of the capsule to vary from their unit values.



$t < 0.0$ $t > 0.0$

Figure 3.4: Tapering effect of $t$ parameter

Having extended the capsule specification thus far, it was observed that the surface edges of a body part were not necessarily aligned with those of the (axis-aligned) bounding box used to determine the geometric transforms applied to its corresponding capsule. Two parameters, $\theta$ and $\phi$, were therefore added to allow the capsule to tilt along a variable axis (Fig. 3.5). Three further parameters ($x_c$, $y_c$, $z_c$) were added to allow the centre of the capsule to be offset from that of the bounding box and, lastly, the tapering parameter $t$ was split in order to allow tapering in two dimensions ($t_x$ and $t_z$).

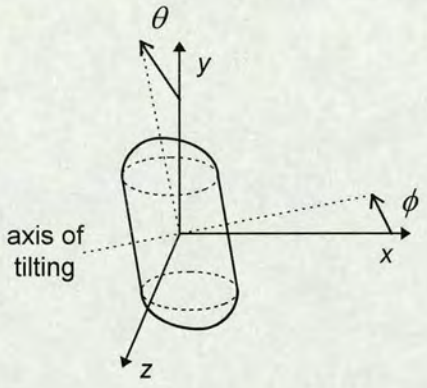



Figure 3.5: Tilting of capsule according to parameters $\theta$ and $\phi$

With each addition of parameters, the accuracy of the best-fitting capsule for each body part was significantly improved. To illustrate this, Table 3.1 details the progressive development of the capsule structure by tabulating the accuracy of the optimum collision object (*i.e.* the final evaluation of the cost function employed by the best-fit algorithm, see below) for each body part of a typical mannequin model,

according to the set of parameters used to specify the capsule structure at that stage in its development.

| **Body Part** (no. of surface sample points) | $r$ | $r_1$ $r_2$ | $r_1$ $r_2$ $t$ | $r_1$ $r_2$ $t$ $x$ $y$ $z$ | $r_1$ $r_2$ $t$ $x$ $y$ $z$ $\theta$ $\phi$ | $r_1$ $r_2$ $t$ $x$ $y$ $z$ $\theta$ $\phi$ $x_c$ $y_c$ $z_c$ | $r_1$ $r_2$ $t_x$ $t_z$ $x$ $y$ $z$ $\theta$ $\phi$ $x_c$ $y_c$ $z_c$ |
|---|---|---|---|---|---|---|---|
| **Chest** (2539) | 9.89857 | 9.8788 | 8.31694 | 7.59283 | 7.56306 | 7.35089 | 6.61597 |
| **Neck** (289) | 0.063132 | 0.060985 | 0.058019 | 0.036929 | 0.0309268 | 0.0309046 | 0.022291 |
| **Head** (944) | 0.875534 | 0.869529 | 0.760079 | 0.51867 | 0.467128 | 0.467388 | 0.366194 |
| **L. Upper Arm** (590) | 1.20962 | 1.20526 | 1.20548 | 0.825487 | 0.342986 | 0.34221 | 0.339447 |
| **L. Lower Arm** (488) | 0.274092 | 0.271445 | 0.269957 | 0.161846 | 0.140195 | 0.140054 | 0.135315 |
| **R. Upper Arm** (609) | 0.748391 | 0.74446 | 0.738617 | 0.578929 | 0.462399 | 0.461746 | 0.459565 |
| **R. Lower Arm** (480) | 0.220836 | 0.220715 | 0.213076 | 0.133054 | 0.126727 | 0.126517 | 0.124043 |
| **Waist** (2085) | 4.56878 | 4.31596 | 3.30774 | 2.99478 | 2.355 | 2.35303 | 2.13899 |
| **L. Upper Leg** (1174) | 2.16089 | 1.97476 | 1.84035 | 1.19843 | 1.17272 | 1.17227 | 1.16001 |
| **L. Lower Leg** (1019) | 1.58194 | 1.57449 | 1.46492 | 0.791371 | 0.743948 | 0.736901 | 0.730975 |
| **R. Upper Leg** (1163) | 2.25969 | 2.12712 | 2.07307 | 1.40972 | 1.35051 | 1.35039 | 1.34623 |
| **R. Lower Leg** (979) | 1.29073 | 1.27397 | 1.1895 | 0.768857 | 0.711752 | 0.705356 | 0.699459 |

Each entry records the final evaluation of the cost function (see Section 3.6) for the best-fitting member of the capsule population. The optimisation software was run on a 175MHz Sun ULTRASparc platform. In each case, the cost of sample points lying outside of the capsule volume was biased by a value of 8.0, and the algorithm was executed until 20 consecutive iterations produced the same minimum cost to an accuracy of 5 decimal places.

Table 3.1: Improvement in capsule fitting afforded by increase in complexity

The final form of the capsule structure, as used by the FIGMENT scheme, is therefore defined by the following 12 parameters:

- $r_1$ and $r_2$ — the rounding factors at the top and bottom of the capsule
- $t_x$ and $t_z$ — the tapering factors in the $x$-dimension and $z$-dimension (the capsule being aligned vertically), where a positive value indicates a widening at the *top* of the capsule

- $x$, $y$, $z$ — the dimensions of the capsule (each of unit value by default)
- $x_c$, $y_c$, $z_c$ — the offset of the centre point of the capsule (each of zero value by default)
- $\theta$ — the angle of tilt of the capsule
- $\phi$ — the angle of rotation (around the $y$-axis) of the tilting axis ($0^\circ$ is $x$-axis, $90^\circ$ is $z$-axis)

This combination of parameters allows a considerable amount of flexibility in the form of the capsule, allowing it to usefully approximate almost every body part of a jointed mannequin model. The upper torso of the female mannequin cannot usually be closely approximated by one capsule alone, but a quite acceptable representation can be obtained by appropriately partitioning the body part into three separate objects.

## 3.6 The genetic best-fitting algorithm

In order to obtain a best-fitting set of capsule objects for a particular mannequin model, an algorithm must be implemented to determine the optimum values of the capsule parameters for each individual body part. As indicated previously, solving the problem of finding the best-fitting capsule for each part using a purely *analytic* method would be unrealistic due to the high number and complex relationships of the 12 parameters. Instead, a form of genetic optimization algorithm has been found to be the best iterative method for the task. Genetic algorithms lend themselves especially well to this type of problem (see Goldberg, 1989) and the particular type of algorithm adopted for the purposes of the FIGMENT scheme is an adaptation of that used by Louchet (1994) and Louchet, Provot and Crochemore (1995) for a different application.

The best-fit algorithm proceeds as follows. For each body part, a 'population' of identical capsule objects (100 to 200 has proved to be a suitable size in practice) is created initially. Each capsule begins with default parameters and is geometrically

transformed such that its bounding box coincides with that of the body part. Each iteration of the optimization then proceeds in four stages:

1. A cost function (detailed below) is evaluated for each capsule to indicate the extent of discrepancy between the surface of that capsule and the corresponding body part.
2. The members of the population are sorted into order of increasing cost.
3. A *mutation* stage occurs during which a certain proportion $p_m$ of the bottom (higher cost) members of the population are replaced with 'mutated' versions of those above (Fig. 3.6). For each new member, a copy of a randomly chosen member from the top $(1-p_m)$ proportion of the population is taken, and each of its parameters are varied randomly within certain limits. The alteration of each parameter is applied according to an approximately normal distribution.
4. A *crossover* stage occurs in which a certain proportion $p_c$ of the members above those replaced during the mutation stage are replaced with 'crossed-over' versions of those above (Fig. 3.6). The new members are created by taking each parameter from a 'parent' chosen randomly from the top $(1-p_m-p_c)$ proportion of the population. In this way, each new member has as many 'parents' as parameters.

The iterations are continued until one of two conditions is fulfilled: either *(1)* the cost of the best-fitting capsule in each generation levels out to within a specified degree of accuracy or *(2)* a specified maximum number of iterations is reached. The algorithm is performed three times for each body part, once for each possible orientation of the initial capsule, and the best fit of the three cases is taken as the final result.

The cost function of the optimization algorithm is implemented as follows. A set of sample points is taken from the surface of the polygonal body part, comprised of the vertices of the polygons (with duplicates being removed) and equally spaced points within the polygon edges (Fig. 3.7). To calculate the cost of a particular capsule, each sample point is subjected to the collision detection routine (described in the following section) in order to compute the nearest point on the surface of the

capsule. The cost is then calculated to be *the sum of the square distances of each sample point from the nearest point on the capsule surface*. This value is therefore a measure of the deviation between the two surfaces and is inversely proportional to the 'fit' of the capsule object in question.

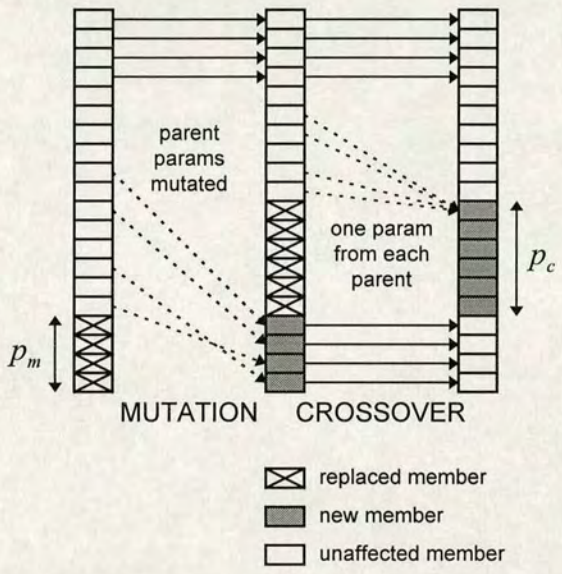



Figure 3.6: Mutation and crossover stages of genetic algorithm

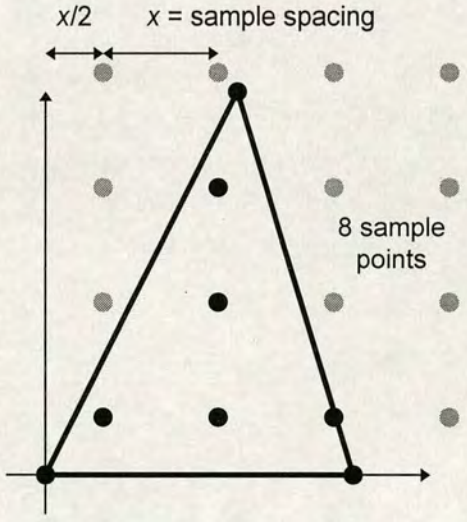



Figure 3.7: Sampling of surface points from polygon

The FIGMENT scheme has obtained excellent results during each stage of the development of the capsule structure (as detailed in the previous section) by using

$p_m = 0.2$ and $p_c = 0.3$ in the algorithm described above with a population of 100 members. Two modifications of the algorithm were made, however, to increase both its speed of convergence and the usability of the best-fitting capsules.

Firstly, the member-choosing function for the *mutation* and *crossover* stages of the genetic algorithm has been biased towards choosing upper members of the population, *i.e.* those with lower cost. This was achieved simply by using a square rather than linear distribution of random choice (Fig. 3.8) and as a result encourages the algorithm to converge to a solution of the desired accuracy in fewer iterations.

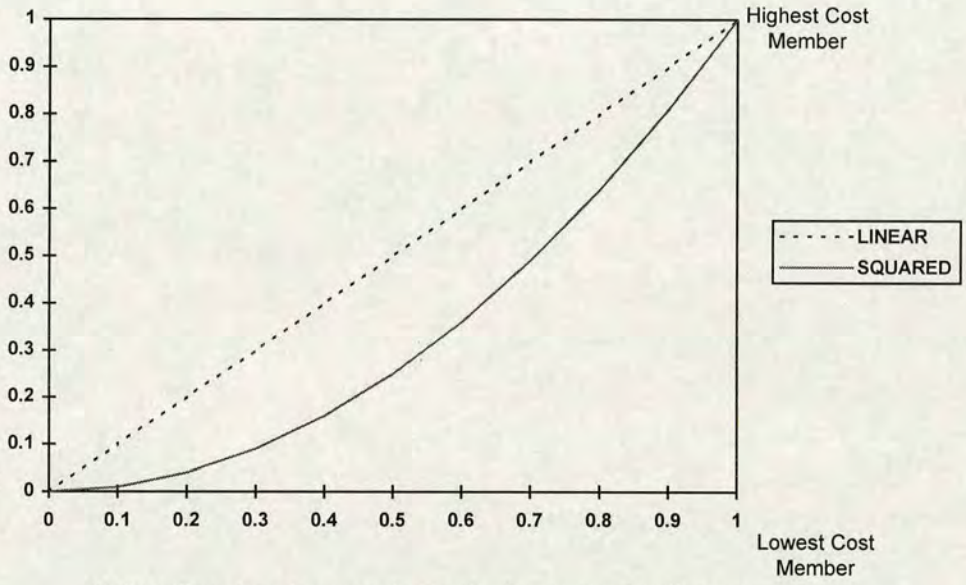



Figure 3.8: Linear vs. squared function for choosing 'parent' member

The second modification was motivated by a consideration of the specific application of the results of the algorithm within the FIGMENT scheme. Since the draping of cloth is more determined by convexities than concavities in the surface of the mannequin model, the cost function for the algorithm has been biased against sample points which lie outside the capsule surface so that resultant capsules tend to *enclose* the original surface rather than not. The biasing of the cost function is implemented by multiplying by a constant value the square distance error for sample points which fall *outside* the capsule volume.

## 3.7 An alternative genetic algorithm

In order to test the efficiency of the original genetic algorithm, an alternative version of the capsule-fitting FIGMENT software was implemented using a second form of genetic algorithm to investigate whether any improvement in processing time could be achieved. The alternative algorithm is analogous to the basic form described by Goldberg (1989) and proceeds in the following manner.

Beginning, as before, with a population of identical capsule members, each iteration consists of five stages:

1. The cost function for each capsule is evaluated in the same way as before.
2. An entirely new 'generation' of members is created which ultimately replace the current population at the end of the iteration.
3. For each member of the new generation, two 'parents' are randomly chosen from the previous generation. A key factor is that the distribution of choices is such that the probability of picking any particular member is *inversely* proportional, in some respect, to its cost.
4. Each parameter of each new member is taken from one or other of the parents (chosen randomly with equal distribution).
5. Each new member is *mutated* in the same way as for the original algorithm with the exception that, in this case, there is only a probability $P_m$ for each individual parameter that it will be altered.

As before, iterations continue until either of the two terminating conditions are met.

Unfortunately, this algorithm proved to perform nowhere near as effectively as the original one—despite prolonged experimentation with varying sizes of population, parent-choosing distributions and mutational probabilities. The convergence of the best-fitting solutions took many more iterations then with the initial algorithm and the minimum cost value for each population often levelled out for periods before descending again, making it difficult to determine when to

terminate the algorithm. For these reasons, the alternative algorithm was discarded in favour of the superior original.

## 3.8 Collision detection and response calculations

The greatest advantage of the capsule approximation method is the low number of calculations required to detect and respond to vertex collisions. By considering the 12 parameters which define the capsule object, it can be observed that all but the rounding and tapering elements correspond to simple geometric transformations and can thus be incorporated into the overall transformation applied to the unit capsule. Determining whether a point $\left(x_p, y_p, z_p\right)$ lies inside the unit capsule defined by the first four parameters ($r_1$, $r_2$, $t_x$, $t_z$) requires a relatively small number of multiplication/addition operations, providing a very fast method of collision detection.

Collision detection

The required calculations proceed as follows: firstly, the inverse transformation matrix is applied to the point, bringing it into the unit space and orientation of the normalized capsule. In addition, the $x_p$ and $z_p$ coordinates of the point are divided by the magnitude of the tapering effect in the *x*- and *z*-dimensions, dependent on the location of the point in the *y*-dimension:

$$\frac{x_p}{1+2t_x\dfrac{y_p}{y}} \Rightarrow x_p \tag{3.1a}$$

$$\frac{z_p}{1+2t_z\dfrac{y_p}{y}} \Rightarrow z_p \tag{3.1b}$$

Hence, $x_p$ and $z_p$ are unaffected if $y_p = 0$ (at the vertical centre) and are multiplied respectively by $(1+t_x)^{-1}$ and $(1+t_z)^{-1}$ if $y_p = \frac{1}{2}y$ (at the top of the capsule). These two transformations effectively bring the point into the local coordinate system of an axis-aligned vertically-oriented unit capsule.

At this point, a check for non-collision is applied; if the absolute value of $x_p$, $y_p$ or $z_p$ exceeds 0.5 then a collision can be immediately ruled out. Otherwise, it must be determined whether the point lies within the top hemispherical, middle cylindrical, or bottom hemispherical parts of the capsule. The distances of the hemispherical sections from the X-Z plane are pre-computed thus:

$$h_1 = 1 - r_1 \tag{3.2a}$$

$$h_2 = r_2 - 1 \tag{3.2b}$$

If $y_p > h_1$ then the point lies in the top hemispherical portion and $y_p$ is normalized further with respect to the vertical scaling of the hemisphere:

$$y'_p = (y_p - h_1)/r_1 \tag{3.3}$$

The distance $d_h$ of the point $\left(x_p, y'_p, z_p\right)$ from the radial centre of the hemisphere is computed in order to test the collision condition:

$$d_h = \sqrt{x_p{}^2 + y'^2_p + z_p{}^2} \tag{3.4}$$

If $d_h > 0.5$ then no collision has occurred; otherwise, collision correction and response computations are performed. The collision detection calculation for the lower hemisphere is identical, substituting $r_2$ and $h_2$ appropriately.

If $y_p$ lies between $h_1$ and $h_2$, *i.e.* in the cylindrical section of the capsule, then the calculations are even simpler. In this case, the distance $d_c$ of the point from the central axis of the cylinder is computed in order to test the collision condition:

$$d_c = \sqrt{x_p{}^2 + z_p{}^2} \tag{3.5}$$

If $d_c > 0.5$ then no collision has occurred.

Collision correction and response

If a collision is detected then both a correction and a corresponding response in the dynamics of the cloth mesh are required. The correction method implemented by the software is that of translating the point immediately to the surface of the capsule; this is required to avoid rendering discrepancies. Although this involves a discontinuity in the dynamics of the mesh, in practice its magnitude is generally small in relation to the sizes of the mesh polygons and any transient mesh distortions of significance are tempered by the instability-countering measures of the FIGMENT physical model (see Section 2.6). In order to compute the response in the dynamics of the mesh, the normal at the surface point of collision is also required. The surface point $(x_s, y_s, z_s)$ and the normal vector $(x_n, y_n, z_n)$ are calculated in each case as follows. For points penetrating the rounded ends of the capsule:

$$(x_s, y_s, z_s) = \left( \frac{1}{2d_h} x_p, \frac{r}{2d_h} y'_p + h, \frac{1}{2d_h} z_p \right) \tag{3.6a}$$

$$(x_n, y_n, z_n) = \left( x_p, \frac{1}{r} y_p, z_p \right) \tag{3.6b}$$

For points penetrating the cylindrical section of the capsule:

$$(x_s, y_s, z_s) = \left( \frac{1}{2d_c} x_p, y_p, \frac{1}{2d_c} z_p \right) \tag{3.7a}$$

$$(x_n, y_n, z_n) = (x_p, 0, z_p) \tag{3.7b}$$

The factors of 2 in the denominators of the above equations are due to the fact that the normalized cylinder and hemispheres have radii of length 0.5 units; the actual simulation code used within an implementation of FIGMENT should normalize for a

unit capsule with dimensions $2^{\times}2^{\times}2$ units in order to reduce the number of arithmetic operations required.

Having obtained the coordinates of the corrected surface point in local space, the point is then transformed back into its original space by reversing the tapering transformations expressed in equations (3.1a) and (3.1b), and then applying the original transformation matrix. The normal vector must also be correspondingly transformed; this is done by applying the rotational part of the original transformation and the *inverse* of the scaling factors. The effect of the tapering transformation on the normal vector is non-trivial but negligible ($t_x$ and $t_z$ are generally of low magnitude) and may therefore be omitted from the calculations. Strictly speaking, if there are unbalanced scaling factors (*e.g.* $s_x \neq s_z$) present in the transformation of the surface point, this point will not always be the closest to the surface with respect to the penetrating point; however, the error is not significant. It should be noted, therefore, that the calculations detailed above provide close *estimations* of the vectors required; the computational cost of implementing more accurate calculations would not be merited by the imperceptible difference in the results.

The determination of the surface point and the normal vector at that point allow a dynamic response to be computed with respect to the colliding node of the cloth mesh. According to the physical model of the cloth, each point in the mesh is attributed a proportion of the masses of the sections with which it is associated (specifically, one third from each section). On collision, the vector dot product of the surface normal vector $\overline{n}$ and the node's velocity $\overline{v}$ is calculated. If this value is positive (*i.e.* the node is travelling away from the surface) then the velocity is left unchanged, otherwise the component of velocity perpendicular to the surface is removed (modelling an inelastic collision):

$$\begin{aligned} \overline{n} \cdot \overline{v} \geq 0 \quad &\Rightarrow \quad \overline{v}' = \overline{v} \\ \overline{n} \cdot \overline{v} < 0 \quad &\Rightarrow \quad \overline{v}' = \overline{v} - (\overline{n} \cdot \overline{v})\overline{n} \end{aligned} \tag{3.8}$$

Within the FIGMENT scheme, collision detection and response is performed at the end of a simulation iteration, following the calculation and application of all other internal and external forces. For any cloth mesh node colliding with the surface of the mannequin, the forces acting on the node perpendicular to the surface are opposed by the solidity of the mannequin body (having a mass far greater than that of the cloth). In the same way as for its velocity, therefore, any component of the node's acceleration $\bar{a}$ acting towards the surface is removed:

$$\begin{aligned} \bar{n}\cdot\bar{a} \geq 0 \quad &\Rightarrow \quad \bar{a}' = \bar{a} \\ \bar{n}\cdot\bar{a} < 0 \quad &\Rightarrow \quad \bar{a}' = \bar{a} - (\bar{n}\cdot\bar{a})\bar{n} \end{aligned} \tag{3.9}$$

However, any force acting on the node towards to the surface will produce an opposing frictional force parallel to the surface (Fig. 3.9) which is proportional in magnitude to the perpendicular component of that acting force (Hannah and Hillier, 1995). If that frictional force exceeds the parallel component of the acting force, then the parallel components of the node's velocity and acceleration are cancelled; this effectively zeroes the node's resultant velocity and acceleration, since the perpendicular components will also have been cancelled by the collision (see above). If the frictional forces does not entirely overcome the acting force, then the node's acceleration is adjusted accordingly. The exact sequence of calculations therefore fall into two cases ($\mu$ is the coefficient of friction between the cloth and mannequin surfaces):

Case 1: $\bar{n}\cdot\bar{a}' \geq 0$ $\quad F_{friction} = 0$

$$\bar{v}'' = \bar{v}' \quad \bar{a}'' = \bar{a}'$$

Case 2: $\bar{n}\cdot\bar{a}' < 0$ $\quad F_{friction} = -\mu\cdot m(\bar{n}\cdot\bar{a}')$

$$F_{friction} \geq m|\bar{a}'| \quad \Rightarrow \quad \bar{v}'' = 0 \quad \bar{a}'' = 0$$

$$F_{friction} < m|\bar{a}'| \quad \Rightarrow \quad \bar{v}'' = \bar{v}' \quad \bar{a}'' = \bar{a}' - \frac{F_{friction}}{m}\cdot\frac{\bar{a}'}{|\bar{a}'|} = \bar{a}'\cdot\left(1 - \frac{F_{friction}}{m|\bar{a}'|}\right)$$

It should be noted that the calculations in the second case can be simplified by removing the term $m$, which corresponds to the mass of the node, thus considering the frictional opposition as an acceleration rather than a force. The resultant velocity and acceleration values ($\bar{v}''$ and $\bar{a}''$) obtained by applying the friction model described above are taken to be the values of $\bar{v}$ and $\bar{a}$ in subsequent calculations concerning node dynamics.

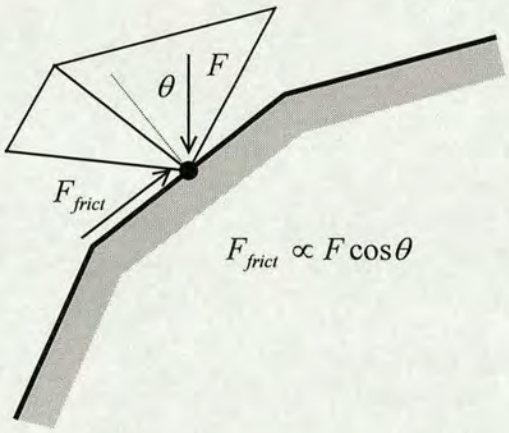



Figure 3.9: Frictional force between cloth node and mannequin surface

Multiple collisions

The set of capsule objects used to represent the collision volume of a mannequin will intersect with each other to some extent. The question then arises as to how to handle the case of a point which penetrates two or more capsules, and a simple algorithm has been incorporated within the FIGMENT scheme to deal appropriately with this situation. During the collision detection phase of an iteration frame, each node of a cloth mesh is tested for collision with each capsule object in an associated list of possible candidates. (Clearly, there is no merit in testing for collision between the nodes of a shirt item and a lower leg object, for example.) If a penetration is detected, the node is translated to the surface of the object and then tested against the remainder of the list. The process is repeated if necessary until no further penetrations are detected and the dynamic collision response is made with respect to the last corrected surface point.

Assuming that the penetration of the capsules is never allowed to occur too deeply (by maintaining appropriate limits on the simulation time-step value), this provides a reliable method of response. Infinite loops will not occur in these cases; indeed, the number of iterations required has never been found to exceeded two in practice.

Although the order in which the approximation objects are tested for collision can affect the collision response, the discrepancy introduced is negligible and does not alter the results in any perceptible way. For example, on examination, the difference between corrected node positions caused by alternative processing orders amounts to less that 1% of the dimensions (*i.e.* edge lengths) of neighbouring mesh sections in the simulations detailed in the following examples.

## 3.9 Speed comparisons

In order to gauge the speed increases afforded by using the capsule method of collision volume approximation, two typical modelling simulations were run on three different platforms—a 200MHz Pentium PC, a 170MHz Sun ULTRASparc and a 180MHz MIPS R5000 Silicon Graphics O2—using an octree-based polygon-to-polygon collision handling algorithm[1] (see Appendix A) in the first instance and the capsule approximation in the second.

Modelling scene *A* consisted of a male mannequin being clothed with a 720-polygon jacket; scene *B* consisted of a male mannequin being clothed with a 1700-polygon sweater and a 1300-polygon pair of trousers; scene *C* consisted of a female mannequin being clothed with a 1100-polygon dress and an 840-polygon jacket. In every case, the simulation was performed for 6.0 'virtual' seconds, with a time-step of 0.001 'virtual' seconds and standard internal force computation methods (see Section 2.3).

[1] As noted in Appendix A, the octree-based algorithm requires the specification of a proximity parameter which determines which set of neighbouring polygons will be tested when establishing the closest correct surface position for a penetrating mesh node. In each of the simulations detailed here, this parameter was set to the *minimum* value which allowed for accurate modelling, *i.e.* avoiding the phenomenon of nodes being 'corrected' to surface positions which significantly distort the mesh.

Table 3.2 indicates the average real time required collision handling during each iteration for both methods, the average percentage of the *total* computation devoted to collision detection for both methods, and the relative speed of the capsule method on the PC, Sun and Silicon Graphics platforms with respect to the octree method.

| Simulation | Average time for collision handling (octree) | Average time for collision handling (capsule) | Average proportion of total computation (octree) | Average proportion of total computation (capsule) | Average speed of capsule method relative to octree method |
|---|---|---|---|---|---|
| Scene *A* (PC) | 140 ms | 14.7 ms | 70.2 % | 25.1 % | 9.5 |
| Scene *A* (Sun) | 160 ms | 50.4 ms | 75.6 % | 59.8 % | 3.2 |
| Scene *A* (SGI) | 261 ms | 38.1 ms | 70.8 % | 33.4 % | 6.9 |
| Scene *B* (PC) | 220 ms | 22.0 ms | 69.6 % | 25.2 % | 10.0 |
| Scene *B* (Sun) | 240 ms | 75.1 ms | 72.6 % | 59.8 % | 3.2 |
| Scene *B* (SGI) | 377 ms | 60.4 ms | 68.3 % | 35.7 % | 6.2 |
| Scene *C* (PC) | 161 ms | 19.0 ms | 73.4 % | 31.6 % | 8.5 |
| Scene *C* (Sun) | 176 ms | 80.4 ms | 77.7 % | 71.4 % | 2.2 |
| Scene *C* (SGI) | 275 ms | 48.1 ms | 73.0 % | 40.1 % | 5.7 |

Table 3.2: Timing results for capsule collision method

The timing data obtained from the three example simulations demonstrate that a significant speed gain can be achieved when using the capsule method rather than the octree method. This gain is most evident on the PC platform; the differences in the results obtained on the three test platforms must be accounted for primarily in terms of the different CPU and maths co-processor architectures.

The speed gain achieved is increased in modelling scenes in which a greater proportion of the cloth sections come into contact with the collision volume (*e.g.* scene *B* compared with scene *C*).[2] Conversely, the speed gain is less pronounced in

[2] All other factors being equal, the gain is also greater in scenes with lower complexity cloth meshes since the larger average area of mesh sections means that a greater number of neighbouring polygons

cases where the ratio of capsule objects to polygons (in the mannequin body) is higher, since the computation time for collision handling is linearly proportional to the number of capsule objects used to represent the polygonal body parts.

## 3.10 Accuracy comparisons

Accuracy analysis[3] for the simulations detailed in the previous section is provided in the following chapter in order to allow simultaneous comparison with the alternative collision volume approximation method described in that chapter. However, the reader may refer in advance to Fig. 4.11, Fig. 4.12 and Fig. 4.13, which plot the mean and standard deviation of the error between corresponding nodes of the cloth meshes in scenes *A*, *B* and *C* (capsule method) compared with scene *A*, *B* and *C* (octree method), and Fig. 4.14(a), Fig. 4.15(a) and Fig. 4.16(a) which plot the distribution of error. In each case, the error (*i.e.* distance between corresponding nodes) is expressed as a percentage of the total height of the mannequin.

## 3.11 Results

To give an indication of the accuracy of the capsule approximation, Fig. 3.10 shows the surface of the capsule representation of the male and female mannequins alongside the mannequins themselves.

Illustrations of the visible differences between the results of the above simulations, in addition to those employing the radial depth method of the following chapter, are provided in Fig. 4.18, Fig. 4.19 and Fig. 4.20.

---

must be considered when correcting the position of penetrating mesh nodes using the octree method (*i.e.* a larger value for the algorithm's proximity parameter).

[3] The reader is referred to Section 2.7 for the details of the accuracy analysis method used here.

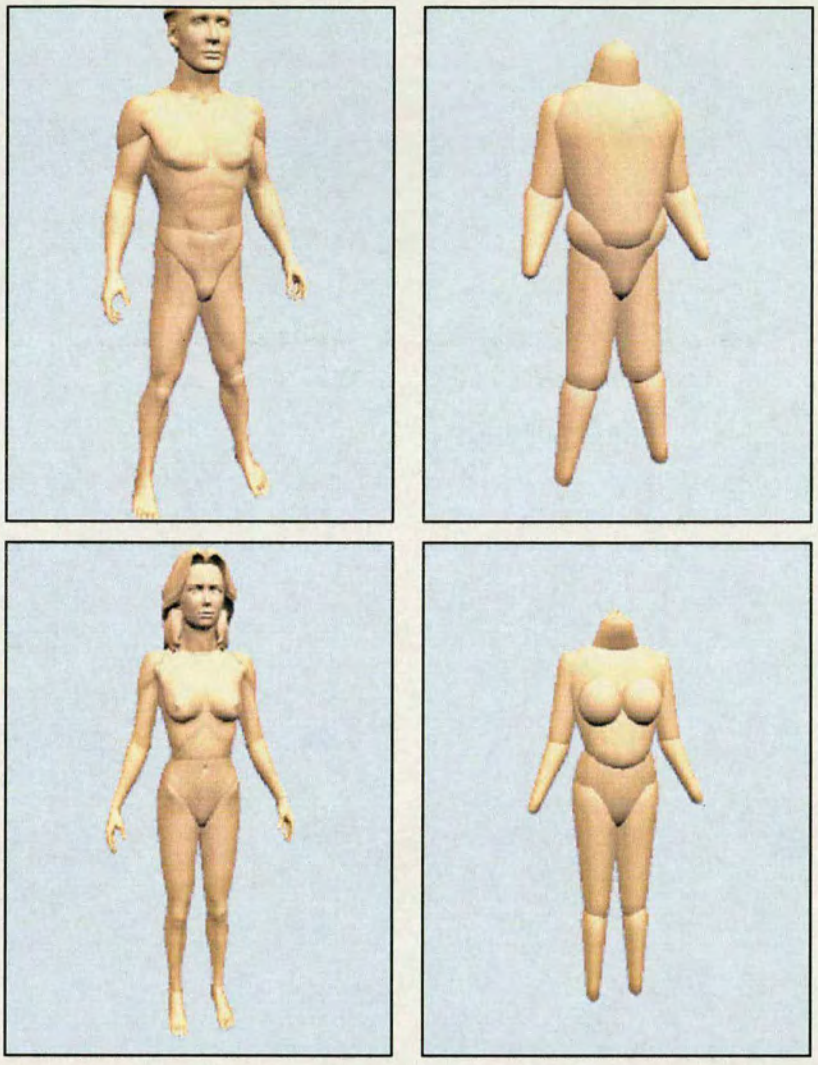


Figure 3.10: Capsule representations of typical mannequins

## 3.12 Conclusions

The results detailed above demonstrate that the capsule volume approximation method provides a highly efficient approach to collision detection and response within the FIGMENT scheme, without abandoning an acceptable level of fidelity in the visual results. Taking only $O(n)$ time to perform, the method provides a substantial speed increase compared to a typical alternative algorithm based on a hierarchical bounding box representation and taking $O(n \log n)$ time. The following advantages of the method should be noted in the context of its implementation within a FIGMENT-based mannequin service:

1. The geometrical simplicity of the approximation structure means that a relatively small number of calculations are required to detect and respond to collisions. In addition, the computation required is *independent* of the complexity (*e.g.* number of polygons) of the represented model and thus no penalty is incurred by using a more highly detailed mannequin model (assuming that the number of capsule objects used to represent the model remains the same).
2. The capsule objects correspond directly to the jointed body parts of the mannequin and thus it is perfectly straightforward to animate the clothed mannequin by applying identical geometric transformations to both the visible body parts and the relevant parameters of the corresponding approximation objects.
3. Similarly, the capsule objects can be geometrically scaled along with the body parts of the mannequin without having to recompute their defining parameters. In this way, a virtual mannequin service can instantly adjust the proportions of the mannequin to match the user's own physique without having to 're-fit' the capsule objects.
4. Deep penetrations of the collision volume are as easily handled as those near the surface. The practical implication of this for cloth modelling is that, in contrast to other modelling systems which will 'draw' the separate panels of cloth together around the mannequin from some initial distance, by using the capsule method

the properly 'seamed' clothing items may simply be superimposed over the mannequin at the outset of the simulation. Areas of cloth which penetrate the surface of the model are immediately 'pushed' to the surface and the overall time for simulation is thus reduced. Such relatively deep penetrations would take a significant number of iterations to be fully corrected, if at all, using a polygon-to-polygon collision detection method.

5. Considerably less information is required to internally represent the shape of the collision structures. In contrast, an efficient polygon-based collision method would most likely pre-compute and store additional information used in collision calculations, *e.g.* normal vectors, as well as that required by hierarchical bounding volume structures.

Although the capsule method (and the radial depth method described in the following chapter) has been specifically developed for use in modelling virtual clothing, the techniques employed suggest further possible application in the area of real-time virtual environments, with respect to both humanoid and non-humanoid avatars.

Finally, the effectiveness of genetic optimization algorithms in determining best-fitting collision structures should also be noted, in an application where analytic or other iterative methods to perform the same function would prove difficult, or impossible, to implement.

## 3.13 Summary

This chapter has introduced the motivation behind the second point of the FIGMENT scheme by highlighting the inappropriateness of previously existing collision detection methods for the application currently under consideration. An alternative approach, collision volume approximation, has been presented as a suitable compromise between the features of the two general categories of collision algorithms, offering efficiency of computation through an acceptable degree of approximation. The chapter concentrated on one form of collision volume approximation, the 'capsule' method, by detailing an algorithm for obtaining a

capsule representation of a mannequin's body and the calculations required to implement collision detection and response using that representation. Finally, timing data, accuracy analysis and visual results were provided from example modelling scenes to demonstrate the speed gains typically obtainable and the corresponding level of error introduced, with respect to an optimised polygon-to-polygon method.

# Chapter 4

# Collision Approximation: Radial Depth Method

## 4.1 Introduction

The previous chapter introduced the problem of collision detection and response in the context of an interactive cloth modelling application, the implementation of which the FIGMENT scheme aims to enable. The approach referred to as 'collision volume approximation' was introduced and justified as a favourable solution. The so-called 'capsule' method of collision volume approximation was presented in detail with timing results illustrating its speed advantage over alternative methods.

This chapter presents an alternative collision volume approximation method, the 'radial depth' method, which may prove more appropriate in FIGMENT-based modelling services which desire (and can allow for) a different emphasis in the conflicting demands for speed and fidelity of results. The 'radial depth' approximation structure is explained in detail, followed by a full account of the algorithms implemented in order to determine the best-fitting set of collision structures for a particular mannequin model.

The calculations required to detect and respond to the collision of a cloth mesh node with a radial depth object are derived in full, before providing an assessment of the speed and accuracy of the implementation of the radial depth method in comparison with both the capsule method and the polygon-based method referred to in the previous chapter. Finally, examples of the visual results obtained when using the radial depth method are provided before drawing conclusions about the relative

merits of the two collision volume approximation methods offered by the FIGMENT scheme.

## 4.2 The 'radial depth' approach

The development of the second collision volume approximation method for the FIGMENT scheme was motivated by a desire to enable a more accurate representation of the surface of a mannequin model whilst maintaining at least the same order of speed improvement provided by such approximation methods. The 'radial depth' method presented here provides a more faithful representation at the price of an approximately doubled computation time for collision detection and response. The choice between the capsule and the radial depth method, for any particular FIGMENT implementation, will depend on the power of the rendering platform, the level of detail required, and even the shape and significance of individual body parts within the mannequin model. Implementations are not committed to using exclusively one or the other, but may combine the advantages of each for an optimum representation and response.

The formulation of this alternative method relied on similar observations regarding the shape of the body parts being approximated. In this case, those observations were that the objects are generally enclosed surfaces and not unduly concave, such that clear cross-sections may be taken along one or more lateral axes. For example, these axes would be approximately vertical for a lower leg object.

If a suitable axis is chosen, a series of cross-sections of the surface of each body part object may be obtained at regular, specified intervals. These intervals do not have to be regularly spaced, although this is the approach taken here; a more sophisticated alternative might compute the points of cross-section to allow for a closer approximation. If the object in question does not exhibit areas of extreme concavity, these cross-sections may be represented as functions of 'radial depth' on a polar coordinate system. Translating these functions to Cartesian coordinates, it may be observed that their periodic nature and generally undulated shape lends itself to Fourier analysis (Fig. 4.1). In the absence of sharp discontinuities (as is found to be

predominantly the case in practice) the function may be approximated using a Fourier series with a relatively low number (5 to 7, say) of harmonics. Thus, the surface of the object can be approximated by specifying the lateral axis and a compact set of floating-point numbers, *i.e.* the Fourier series coefficients for each sample of cross-section taken.

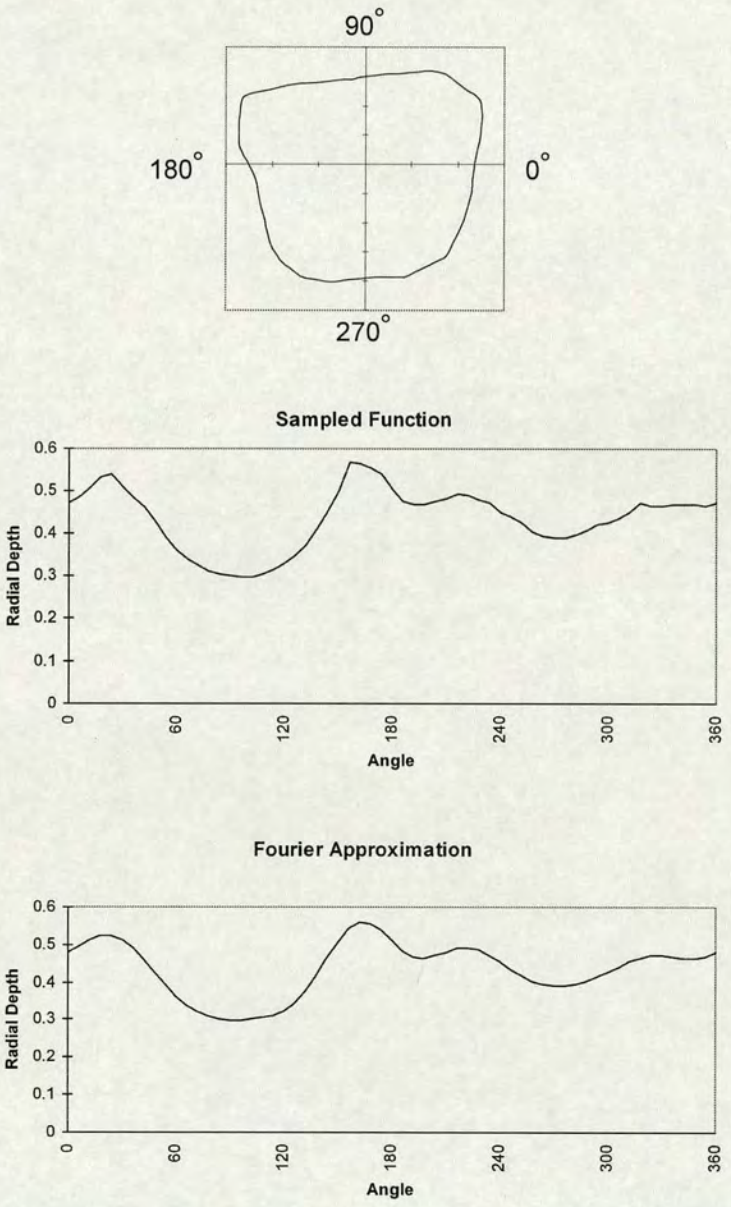



Figure 4.1: Typical cross-section of mannequin body part with corresponding sampled 'radial depth' function and Fourier series approximation

The increase in the accuracy of the surface representation inevitably leads to an increase in the amount of information required to define the surface (as compared to the capsule method). However, if a reasonable number of cross-sections are taken (*e.g.*, 20) then there is still a considerable reduction in comparison with an explicit definition of the model surface (including typically pre-calculated information such as normal vectors).

A question might be raised at this point: why use a Fourier approximation at all, rather than retaining a dense sampling of surface points? Firstly, the Fourier function offers a smoother representation of the body surface which is more in line with a 'real life' body surface. This could be matched by a high density set of sample points, although interpolation calculations would still be required in place of the trigonometric functions. Secondly, with a large set of data points being continually referenced, memory may or may not be an issue. If data requires downloading, this is an additional factor to be considered. However, neither of these points necessitates the use of a Fourier representation, and an implementation which omits this feature would be equally valid. In fact, where resources allow, a combination of methods may prove to be an optimum solution: a Fourier representation can be obtained and used for compact storage, while a specified number of sample points can be precomputed by the modelling software for each cross-section to be subsequently used (with interpolation) for collision detection, thus avoiding the use of sine-cosine calculations during simulation. Timing data for an implementation of this method is provided at the end of the chapter for comparison purposes.

## 4.3 The best-fitting algorithm

As stated previously, the definition of a radial depth object includes both the lateral axis from which the cross-sections are taken and a list of Fourier coefficients for each cross-section. If $N$ cross-sections are taken from each object, and Fourier analysis is performed for $M$ harmonics, then the total number of specified coefficients will be $N(2M+1)$. Also required will be a geometrical transformation matrix which corresponds to the translation of local object space (in which the lateral

axis is the vertical axis and the origin is the centre point of that axis) into global space, *i.e.* that of the modelling scene.

A combination of analytic and iterative methods are employed in order to find the best-fitting radial depth object for a particular body part. The algorithm proceeds in three distinct stages:

1. The optimum lateral axis for taking cross-sections is determined.
2. A set of radial depth values is computed for each cross-section taken at regular intervals along the lateral axis.
3. Fourier analysis is applied to each set of radial depth values and any subsequent adjustment of the resulting Fourier coefficients is performed.

Determining the lateral axis

In order to obtain the most appropriate lateral axis from which to take cross-sections, the FIGMENT scheme employs a genetic search algorithm similar in form to that described in the previous chapter for determining best-fitting capsule objects. The axis, being a line in three-dimensional space specified by a position vector and a direction vector, is defined by six floating-point parameters. An initial 'population' of potential axes is therefore created, consisting of identical members corresponding to an axis which passes through the centre of the bounding box of the body part and is aligned with the longest dimension of the bounding box.

The search algorithm proceeds in a similar fashion to that detailed previously, performing mutation and cross-over variations in each generation. The cost function for any particular axis is evaluated by computing the extent to which each polygon of the body part faces away from the axis and averaging over all the polygons. Referring to Fig. 4.2, the cost function is computed as follows:

$$F_{cost}(axis) = -\frac{1}{n}\sum_{i=1}^{n}\frac{\overline{n}_i \cdot \overline{v}_i}{|\overline{v}_i|} \qquad (4.1)$$

where $n$ is the number of polygons, $\overline{n}_i$ is the normal vector of the $i^{th}$ polygon, and $\overline{v}_i$ is the shortest direction vector from the centre of the $i^{th}$ polygon to the axis in question. As before, the algorithm iterates until the cost of the optimum axis levels out to a specified degree or until a maximum number of iterations is reached.

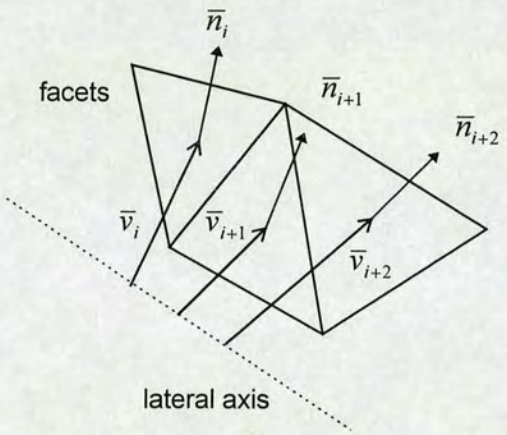



Figure 4.2: Computation of cost function for lateral axis

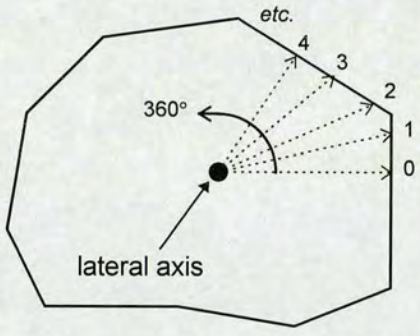



Figure 4.3: Sampling of a cross-section taken from the polygonal object

Determining the radial depth values

Having determined the optimum lateral axis for taking cross-sections, a set of radial depth values for each cross-section is obtained as follows. A line perpendicular to the axis is taken and rotated $360^{\circ}$ around the axis at regular intervals. At each interval, the points at which the line intersects the polygons of the body part are determined, and the radial depth value is taken to be the distance of the furthest point from the axis (Fig. 4.3). In the majority of cases, there will only be one polygon

which is intersected by each line, although in other cases the furthest point is used since that point corresponds to the outermost surface of the body part.

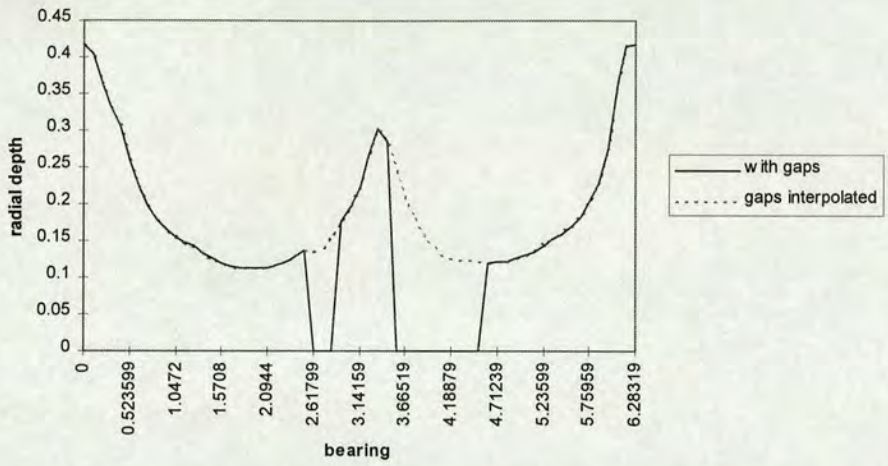



Figure 4.4: Radial depth function with gaps and with interpolation

Determining the Fourier coefficients

Before applying Fourier analysis to the set of radial depth values obtained previously, any harsh discontinuities in the sampled radial depth function are removed by smoothly interpolating 'gaps'. This is necessary in order to avoid 'ripples' in the Fourier approximation and thus a poor representation of the mannequin surface. The gaps themselves are caused by an absence of polygons around the lateral axis at particular points; since these spaces are not determinative of the drape of the clothing over the mannequin, it is quite acceptable to 'fill them in'. Fig. 4.4 shows a typical radial depth function with gaps, and its corresponding Fourier approximation, compared with the same function having had its gaps interpolated. The method of interpolation used here is to insert a sinusoidal function with a magnitude proportional to the width of the gap. A gap extending the full period of the function (which, of course, never actually occurs) would have a minimum point which touches the horizontal axis; a gap extending half the length of the function would have a minimum point halfway between its maxima and the horizontal axis. The reason for choosing a sinusoidal interpolation is that it allows the

Fourier approximation to more closely follow the significant proportions of the radial depth function.

After interpolating the gaps, a Fourier analysis is applied to the radial depth function for a specified harmonic resolution. The resultant Fourier coefficients allow an approximation of the original cross-section to be obtained. However, as is the case with the capsule method of approximation, it is appropriate for the approximation structure to enclose more of the original surface than not; yet the Fourier analysis alone results in an equal proportion of enclosed and protruding surface (Fig. 4.5).

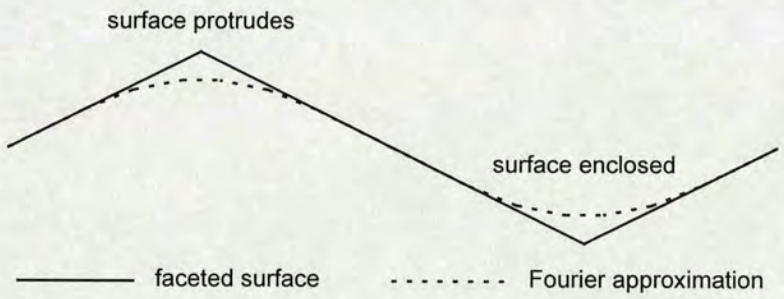



Figure 4.5: Protrusion and enclosure of original surface in equal proportions

In order to accomplish a more favourable enclosure of the original surface, another genetic optimization algorithm is therefore applied to the coefficients of the Fourier approximation function. In this case, each member of the population[1] consists of a set of Fourier coefficients, and the cost function for each member is evaluated as the root-mean-square of the error between the radial depth $D_{radial}(\theta)$ of each of the $N$ sample points (from the original cross-section) and its approximated value according to that set of coefficients:

$$F_{cost}(m) = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1}\left[D_{radial}\left(\frac{2\pi n}{N}\right) - D_{approx}\left(\frac{2\pi n}{N}\right)\right]^2} \tag{4.2}$$

where

$$D_{approx}(\theta) = A_0 + \sum_{n=1}^{M} A_n \cos n\theta + \sum_{n=1}^{M} B_n \sin n\theta \tag{4.3}$$

[1] A population size of 200 has proved to be effective in practice.

and $\{A_0, A_1, \ldots A_M, B_1, \ldots B_M\}$ are the Fourier coefficients for member $m$.

As it stands, the cost function would evaluate to zero initially for each member, since the Fourier approximation would be equally balanced between positive and negative errors, but by weighting the *negative* error values the optimization algorithm provides a set of adjusted coefficients for an approximating function which will enclose more of the original cross-section proportional to that weighting value.

## 4.4 Collision detection and response calculations

As with the capsule structure, the great advantage of the radial depth structure is the relatively low number of calculations required to determine whether a vertex has penetrated a particular radial depth object. For detecting the collision of $n$ nodes of a cloth mesh with a body part of $m$ polygons, the algorithm performs in $O(n)$ time.

### Collision detection

For determining whether a particular vertex $(x_p, y_p, z_p)$ penetrates a radial depth object, the required calculations proceed as follows. Firstly, an inverse transformation matrix is applied to the vertex, bringing it into the local coordinate system of the object such that the vertical axis corresponds to the lateral axis of the object. For every vertex not eliminated by a bounding-box test, the $y$-coordinate is examined to determine between which two cross-sections the vertex lies (Fig. 4.6). The bearing $\theta$ of the vertex around the $y$-axis is calculated (where $0^\circ$ corresponds to the $x$-axis), and the radial depth at that angle for each cross-section is obtained by evaluating the Fourier functions (*cf.* equation (4.3)) for each of the two sets of coefficients, giving values $d_1$ and $d_2$. The radial depth of the surface of the object, with respect to the vertex, is computed by interpolating the values of $d_1$ and $d_2$. (In Fig. 4.6, the radial depth is $\frac{1}{2}d_1 + \frac{1}{2}d_2$ since the vertex lies exactly between the two cross-sections.) The collision condition is fulfilled if the distance of the vertex from the $y$-axis is less than or equal to the corresponding radial depth value.

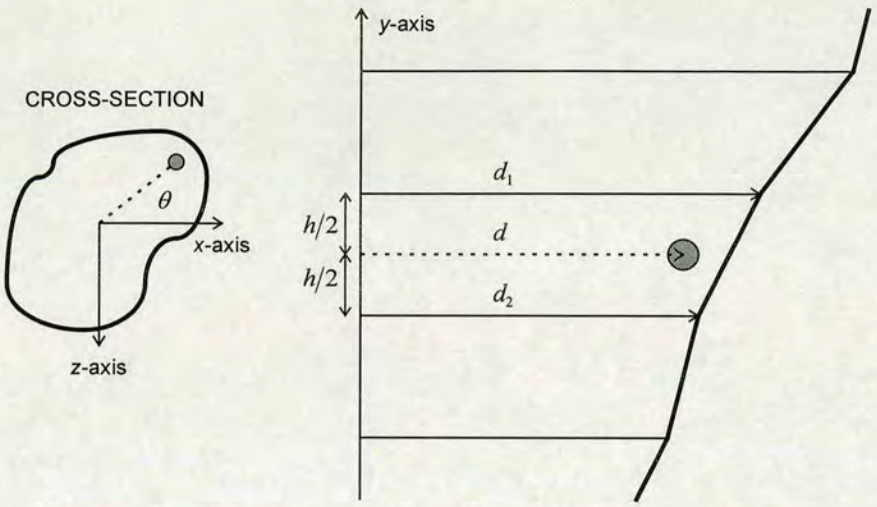



Figure 4.6: Evaluation of the collision condition for a radial depth object

Collision correction and response

As before, on detecting a collision, both a correction and a corresponding response in the dynamics of the cloth mesh are required. The vertex is translated to the surface of the object at the point $(x_s, y_s, z_s)$. The simplest way to locate this point is by 'pushing' the vertex out perpendicularly to the *y*-axis until it meets the surface:

$$(x_s, y_s, z_s) = \left( \frac{d_{radial}}{d_p} x_p, y_p, \frac{d_{radial}}{d_p} z_p \right) \tag{4.4}$$

where $d_p = \sqrt{x_p^{\,2} + z_p^{\,2}}$ and $d_{radial}$ is the corresponding radial depth.

Computationally, this is a highly efficient method of correcting the vertex position and works well for cases in which the difference $\Delta d$ between the radial depths of the two surrounding cross-sections, $d_1$ and $d_2$, is less than the distance between those cross-sections (Fig. 4.7). However, for penetrations where this is not the case—typically occurring at the two ends of the radial depth object—this correction method can result in large and inaccurate displacements of the vertex (Fig. 4.8) leading to undesirable distortions in the cloth mesh. This can be avoided by translating the vertex to the *nearest* point on the line segment considered between the

surface points $A$ and $B$ taken from the two surrounding cross-sections (Fig. 4.9). The position vector $\bar{s}$ of the nearest surface point $S$ can be calculated as follows (where $\bar{a}$ and $\bar{b}$ are the position vectors of points $A$ and $B$ in Fig. 4.9):

$$\bar{s} = \bar{a} + k(\bar{b} - \bar{a}) \tag{4.5}$$

The line segment $\overline{SP}$ must be perpendicular to $\overline{AB}$, therefore:

$$(\bar{p} - \bar{s}) \cdot (\bar{b} - \bar{a}) = 0 \tag{4.6}$$

Substituting (4.5) into (4.6):

$$(\bar{p} - \bar{a} - k(\bar{b} - \bar{a})) \cdot (\bar{b} - \bar{a}) = 0$$

$$\Rightarrow \quad (\bar{p} - \bar{a}) \cdot (\bar{b} - \bar{a}) - k(\bar{b} - \bar{a}) \cdot (\bar{b} - \bar{a}) = 0$$

$$\Rightarrow \quad k = \frac{(\bar{p} - \bar{a}) \cdot (\bar{b} - \bar{a})}{(\bar{b} - \bar{a}) \cdot (\bar{b} - \bar{a})} \tag{4.7}$$

Since the point $S$ must lie between $A$ and $B$, the value of $k$ is subsequently limited to $0 \le k \le 1$.

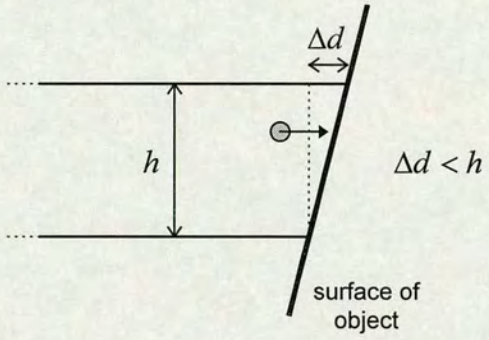



Figure 4.7: Condition for which simple correction method is appropriate

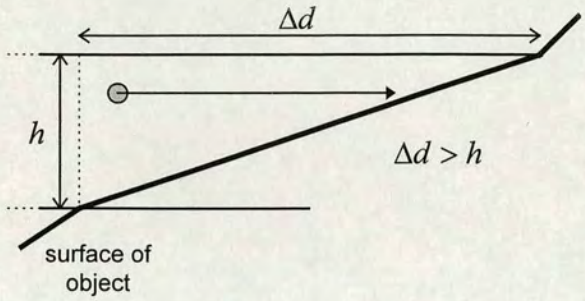



Figure 4.8: Condition for which simple correction method is inappropriate

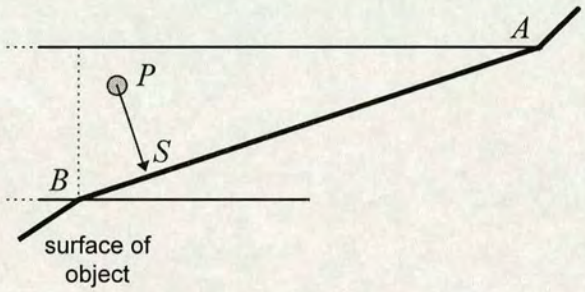



Figure 4.9: Penetrating node moved to nearest point on surface

An estimate of the surface normal vector $(x_n, y_n, z_n)$ at $S$ may be obtained by considering the section of the radial depth object as if it were part of a cone with a slope equivalent to that of the line segment $\overline{AB}$ (Fig. 4.10). The *x*- and *z*-components of the normal vector are therefore proportional to those of the position vector of the surface point, and the *y*-component is inversely proportional to the slope of the surface:

$$x_n \propto x_s \tag{4.8a}$$

$$y_n \propto \frac{d_1 - d_2}{h} \tag{4.8b}$$

$$z_n \propto z_s \tag{4.8c}$$

$$\sqrt{x_n^{\,2} + y_n^{\,2} + z_n^{\,2}} = 1 \tag{4.9}$$

From (4.8a), (4.8b), (4.8c) and (4.9):

$$(x_n, y_n, z_n) = k\left(x_s, \sqrt{x_s^2 + z_s^2}\,\frac{d_1 - d_2}{h}, z_s\right) \tag{4.10}$$

$$\text{where } k = \left|\left(x_s, \sqrt{x_s^2 + z_s^2}\,\frac{d_1 - d_2}{h}, z_s\right)\right|^{-1}$$

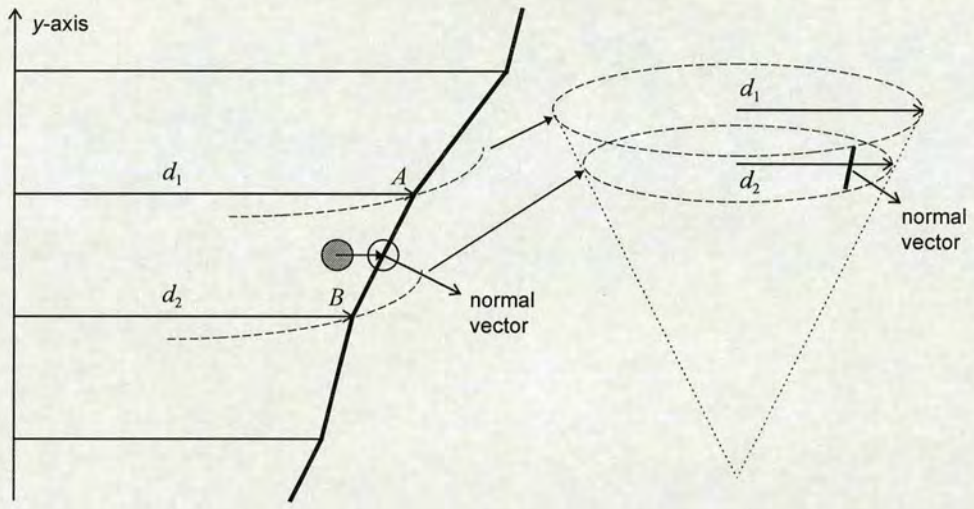



Figure 4.10: Calculation of surface normal vector from cone approximation

A 'purer' calculation of the surface normal vector would take fully into account the *gradient* of the radial depth function (assumed to be zero in the method of approximation given above) which is the derivative function of the Fourier series:

$$D'_{approx}(\theta) = -\sum_{n=1}^{M} A_n \sin n\theta + \sum_{n=1}^{M} B_n \cos n\theta \tag{4.11}$$

To do this would be time-consuming and counterproductive, however, since this increase in accuracy makes little difference to the overall dynamics of the simulated cloth as it interacts with the surface of the mannequin. (Refer to Appendix B for the calculations involved.)

The resultant values of the corrected surface position of the vertex and the surface normal vector must finally be transformed back from the local coordinate system to global space. Having obtained these values, the corresponding dynamic response in

the nodes of the cloth mesh is computed in the same way as for the capsule method (refer to Section 3.8).

## 4.5 Speed comparisons

As described in the previous chapter, in order to gauge the speed increases afforded by using the radial depth method of collision volume approximation, two typical modelling simulations were run on three different platforms—a 200MHz Pentium PC, a 170MHz Sun ULTRASparc and a 180MHz MIPS R5000 Silicon Graphics O2—using an octree-based polygon-to-polygon collision handling algorithm[2] (see Appendix A) in the first instance and the radial depth approximation in the second. In these simulations, the radial depth representation of the mannequin was obtained using Fourier analysis to the *seventh* harmonic, *i.e.* each cross-section being defined by 15 coefficients.

Modelling scene *A* consisted of a male mannequin being clothed with a 720-polygon jacket; scene *B* consisted of a male mannequin being clothed with a 1700-polygon sweater and a 1300-polygon pair of trousers; scene *C* consisted of a female mannequin being clothed with a 1100-polygon dress and an 840-polygon jacket. In every case, the simulation was performed for 6.0 'virtual' seconds, with a time-step of 0.001 'virtual' seconds and standard internal force computation methods (see Section 2.3).

Table 4.1, which should be referred to in conjunction with Table 3.2, indicates the average real time required collision handling during each iteration for the radial depth method, the average percentage of the *total* computation devoted to collision detection for that method, and the relative speed of the radial method on the PC, Sun and SGI platforms with respect to *both* the octree method (see Appendix A) and the capsule method.

[2] As for the simulations detailed in the previous chapter, the proximity parameter for the octree algorithm was set to the *minimum* value which allowed for accurate modelling, *i.e.* avoiding the phenomenon of nodes being 'corrected' to surface positions which significantly distort the mesh.

| Simulation | Average time for collision handling (radial depth) | Average proportion of total computation (radial depth) | Average speed of radial depth method relative to octree method | Average speed of radial depth method relative to capsule method |
|---|---|---|---|---|
| Scene *A* (PC) | 25.7 ms | 36.8 % | 5.4 | 0.57 |
| Scene *A* (Sun) | 92.0 ms | 73.2 % | 1.7 | 0.55 |
| Scene *A* (SGI) | 67.0 ms | 47.6 % | 3.9 | 0.57 |
| Scene *B* (PC) | 39.1 ms | 37.4 % | 5.6 | 0.56 |
| Scene *B* (Sun) | 140 ms | 73.5 % | 1.7 | 0.54 |
| Scene *B* (SGI) | 102 ms | 48.6 % | 3.7 | 0.59 |
| Scene *C* (PC) | 24.2 ms | 36.9 % | 6.7 | 0.79 |
| Scene *C* (Sun) | 89.8 ms | 73.4 % | 2.0 | 0.90 |
| Scene *C* (SGI) | 63.4 ms | 47.4 % | 4.3 | 0.76 |

Table 4.1: Timing results for radial depth collision method

The timing data obtained from the three example simulations demonstrate that a significant speed gain can be achieved when using the radial depth method rather than the octree method. As with the capsule method, this gain is most evident on the PC platform.

In the examples given, the radial depth method of collision handling operates at approximately one-half the speed of the capsule method for the male mannequin and approximately three-quarters the speed of the capsule method for the female mannequin. This difference is accounted for primarily by the fact that the radial depth approximation of the female chest body part requires only one collision object whereas the capsule approximation of the same body part requires three collision objects (due to the shape of the bust). Thus, this is one example of how a radial depth representation (at least, in part) may be preferable to the equivalent capsule representation.

It was noted above (Section 4.2) that where memory resources allow, an even better time response can be achieved by implementing a certain degree of precomputation. Rather than computing Fourier functions 'on the fly', or even using

look-up tables for sine and cosine values, a set of sample depths can be evaluated for each cross-section of each radial depth object. When determining the radial depth of the object surface with respect to a penetrating cloth mesh node, two of the sampled values are interpolated to obtain an estimated radial depth at the bearing $\theta$ of the node around the local $y$-axis of the object.

Scenes $A$, $B$ and $C$ were simulated using this method and the timing results (obtained on the 200MHz Pentium PC platform) are shown in Table 4.2. The table also indicates the speed advantage with respect to the original implementation of the radial depth method (which evaluates pure Fourier functions) and the implementations of the octree and capsule collision methods. The results show a consistent 30% speed increase in the time devoted to collision handling when using the precomputation implementation. It will also be noted that for scene $C$, this method practically equalled that of the equivalent capsule method, which required more volume approximation objects for an adequate representation of the mannequin.

| Simulation | Average time for radial depth collision handling (pure Fourier) | Average time for radial depth collision handling (sampled) | Average speed of sampled method relative to pure Fourier method | Average speed of radial depth (sampled) method relative to octree method | Average speed of radial depth (sampled) method relative to capsule method |
|---|---|---|---|---|---|
| Scene $A$ (PC) | 25.7 ms | 19.8 ms | 1.3 | 7.1 | 0.74 |
| Scene $B$ (PC) | 39.1 ms | 30.1 ms | 1.3 | 7.3 | 0.73 |
| Scene $C$ (PC) | 24.2 ms | 19.1 ms | 1.3 | 8.4 | 0.99 |

Table 4.2: Timing results for radial depth collision method with precomputation

## 4.6 Accuracy comparisons

In this section, accuracy analysis[3] is provided for the simulations detailed in the previous section and also for those detailed in the previous chapter (Sections 3.9 and 3.10). Fig. 4.11 plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes in scene $A$ (capsule method) and scene $A$ (radial depth method) compared with scene $A$ (octree method). In the same way, Fig.

[3] The reader is referred to Section 2.7 for the details of the accuracy analysis method used here.

4.12 plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes in scene *B* (capsule method) and scene *B* (radial depth method) compared with scene *B* (octree method), and Fig. 4.13 plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes in scene *C* (capsule method) and scene *C* (radial depth method) compared with scene *C* (octree method). In each case, the error (*i.e.* distance between corresponding nodes) is expressed as a percentage of the total height of the mannequin.

Fig. 4.14 plots the *distribution* of error between corresponding nodes (a) for scene *A* (capsule method) compared with scene *A* (octree method) and (b) for scene *A* (radial depth method) compared with scene *A* (octree method). In the same way, Fig. 4.15 and Fig. 4.16 plot the distribution of error for scene *B* and scene *C*, respectively. As before, the error is expressed as a percentage of the total height of the mannequin.

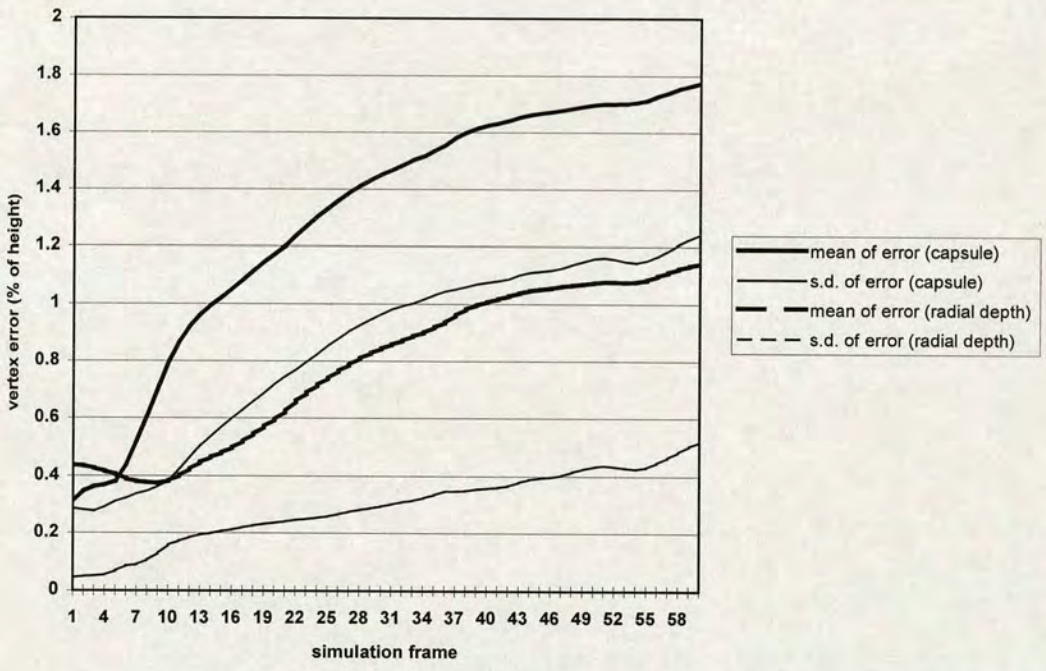



Figure 4.11: Accuracy of simulation relative to octree method (scene *A*)

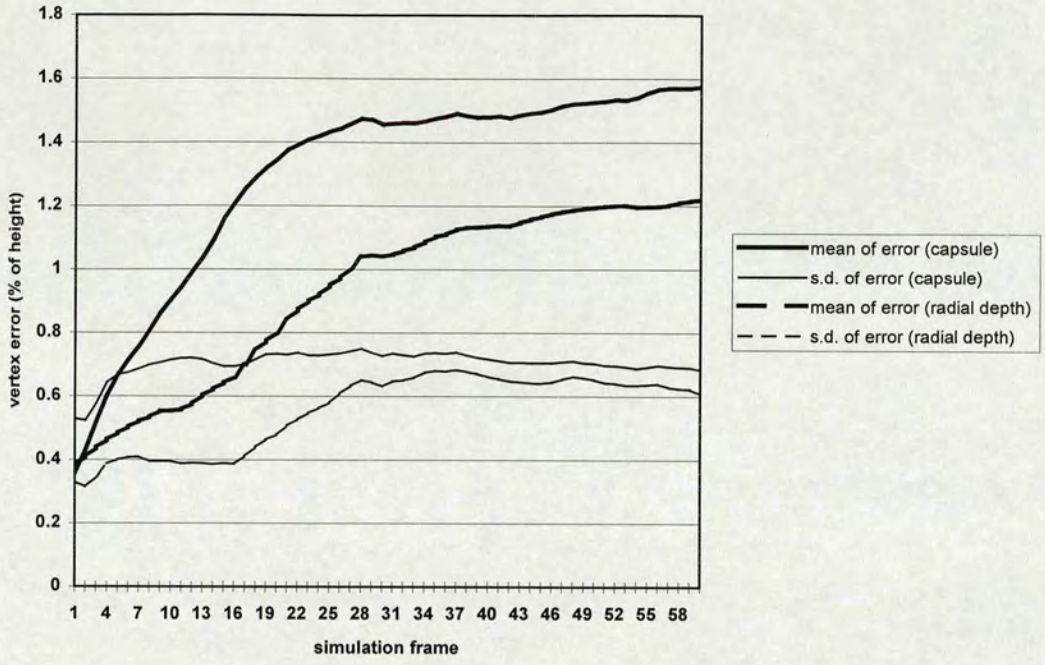



Figure 4.12: Accuracy of simulation relative to octree method (scene *B*)

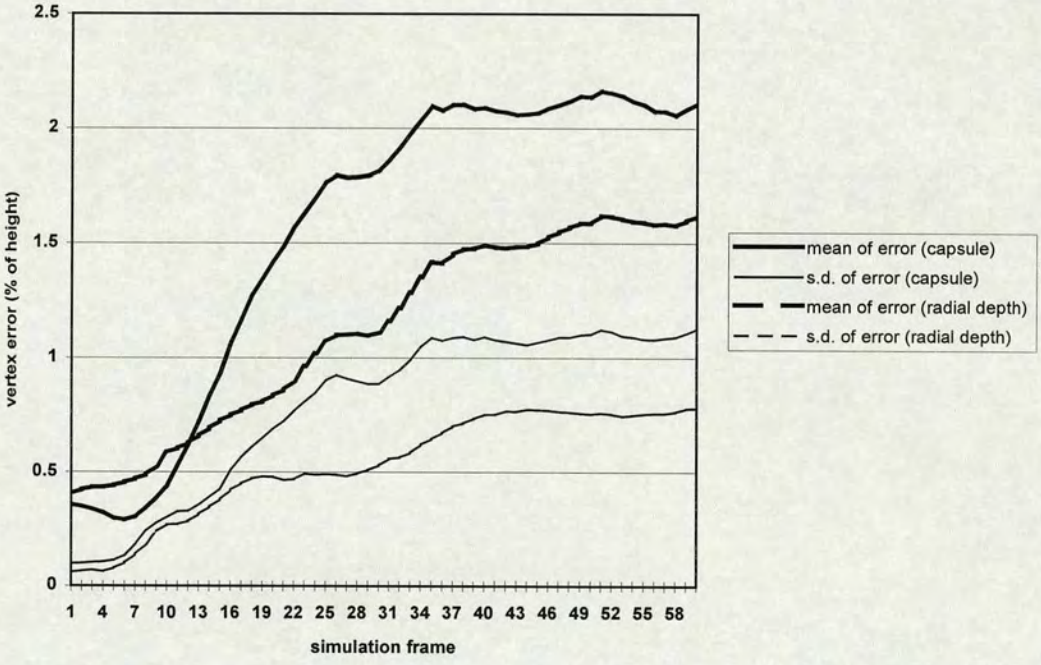



Figure 4.13: Accuracy of simulation relative to octree method (scene *C*)

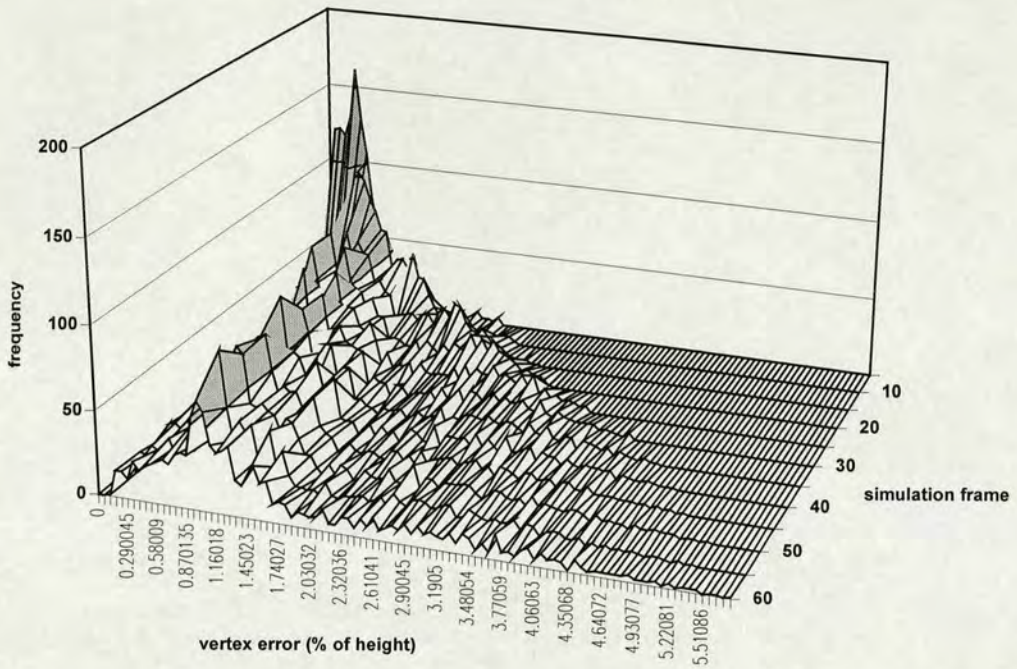



Figure 4.14(a): Distribution of error (capsule) relative to octree method (scene *A*)

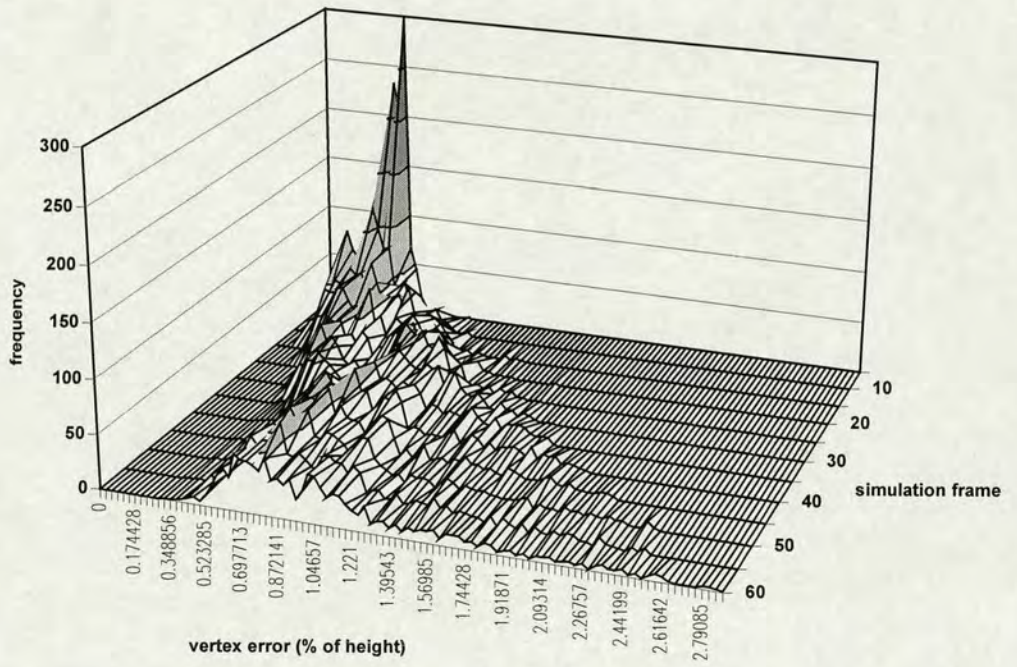



Figure 4.14(b): Distribution of error (radial depth) relative to octree method (scene *A*)

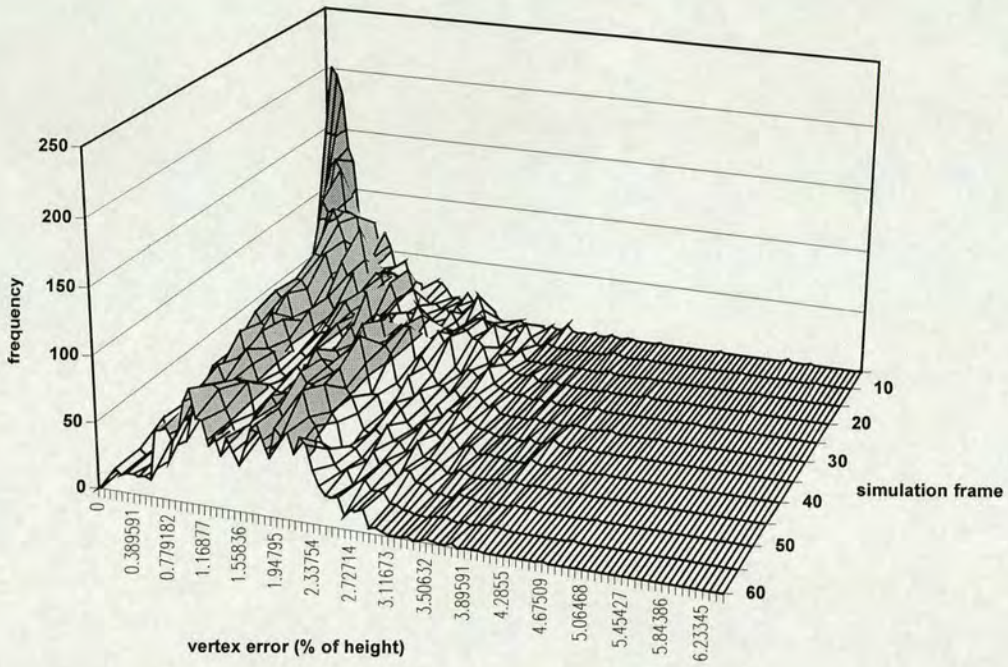



Figure 4.15(a): Distribution of error (capsule) relative to octree method (scene *B*)

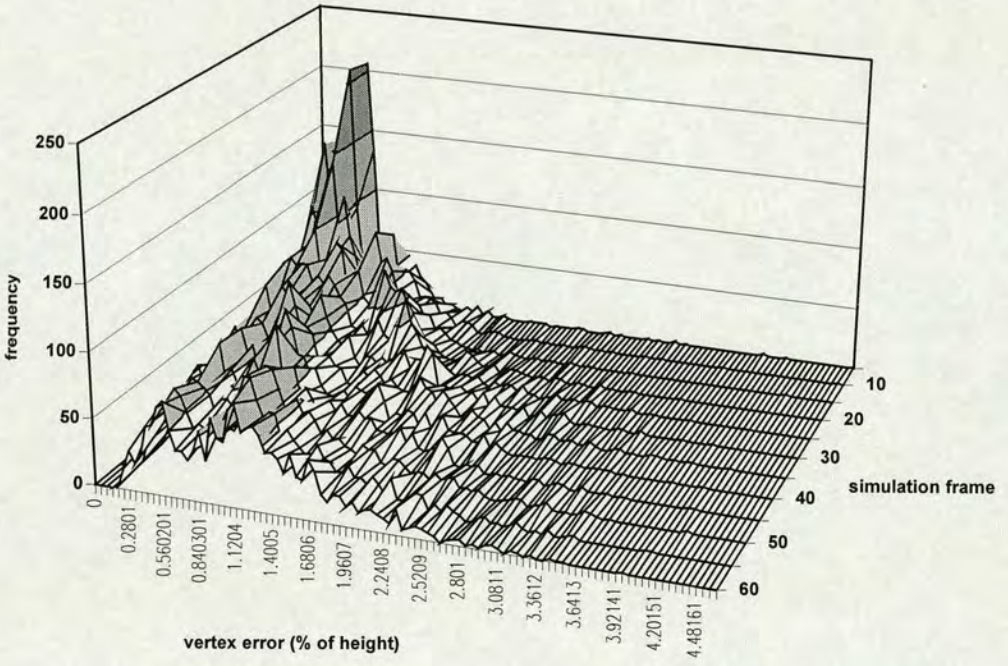



Figure 4.15(b): Distribution of error (radial depth) relative to octree method (scene *B*)

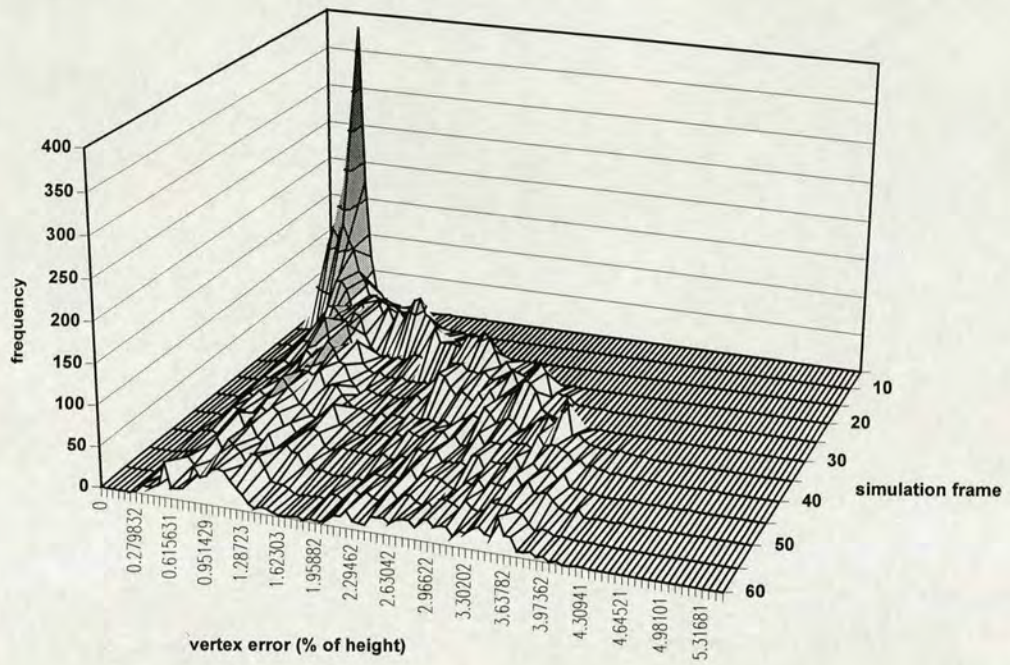



Figure 4.16(a): Distribution of error (capsule) relative to octree method (scene *C*)

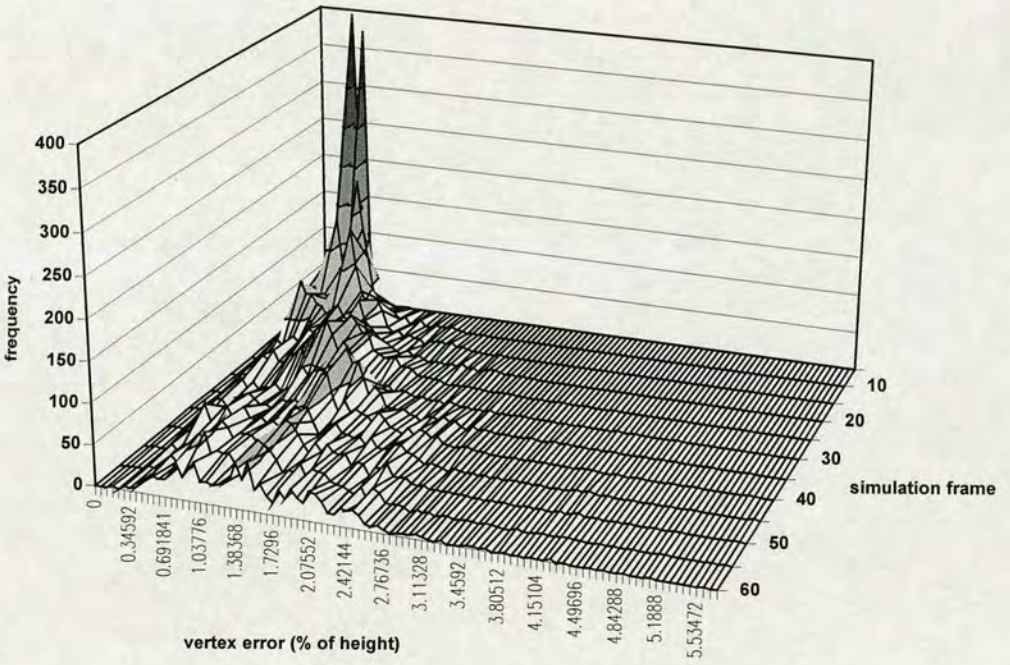



Figure 4.16(b): Distribution of error (radial depth) relative to octree method (scene *C*)

## 4.7 Results

To give an indication of the accuracy of the radial depth approximation, Fig. 4.17 shows the surface of the radial depth representation of the male and female mannequins alongside the mannequins themselves.

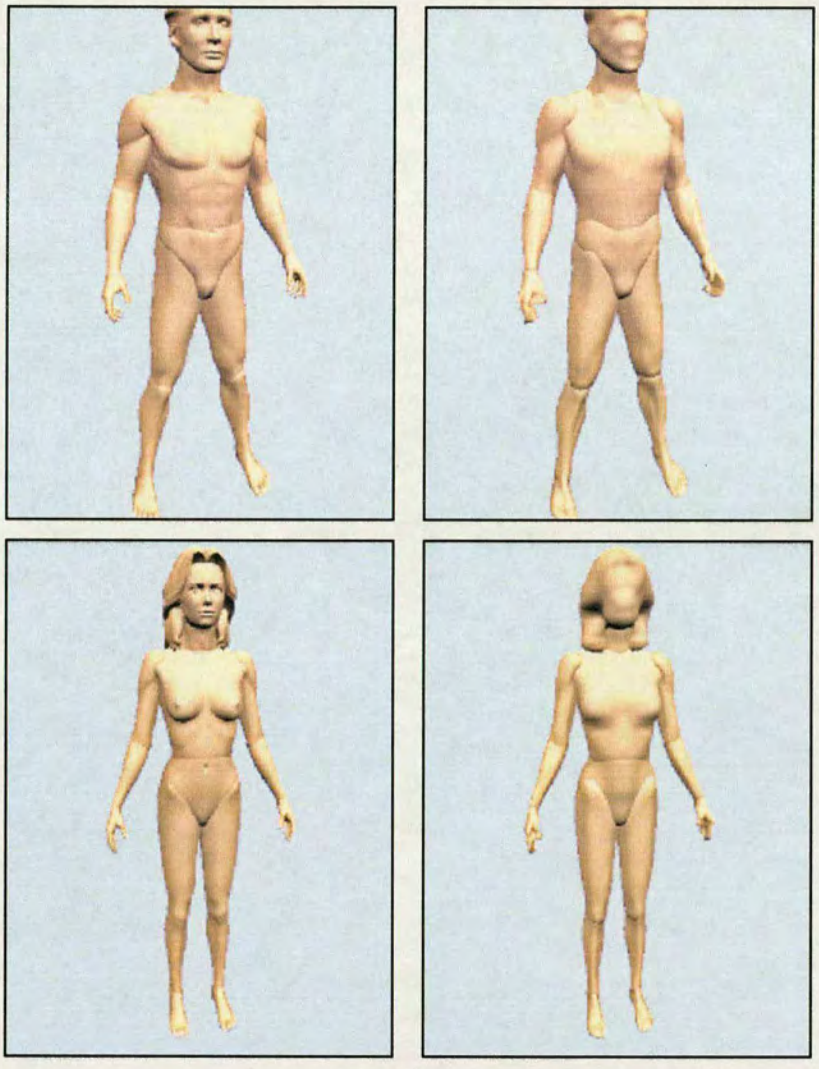


Figure 4.17: Radial depth representations of typical mannequins

To give an indication of the visible differences between the results of the example simulations, Fig. 4.18 shows the final frames of the simulations for scene *A* (octree method), scene *A* (radial depth method) and scene *A* (capsule method), Fig. 4.19 shows the final frames of the simulations for scene *B* (octree method), scene *B* (radial depth method) and scene *B* (capsule method), and Fig. 4.20 shows the final frames of the simulations for scene *C* (octree method), scene *C* (radial depth method) and scene *C* (capsule method).

## 4.8 Conclusions

The results detailed above demonstrate that the radial depth volume approximation method also provides a highly efficient approach to collision detection and response within the FIGMENT scheme, whilst maintaining an acceptable level of fidelity in the visual results. The accuracy analysis demonstrates that the method provides a greater degree of accuracy in practice when compared to the capsule method; furthermore, it can be clearly seen from Fig. 4.17 how closely the surface of the mannequin can be approximated using radial depth objects. Although the computation required for handling collisions with respect to a radial depth approximation is greater than that for a capsule approximation, the former method still provides a substantial speed increase compared to a typical alternative algorithm based on a hierarchical bounding box representation. It should also be noted that the five additional advantages of the capsule method, listed in Section 3.12, also apply to the radial depth method.

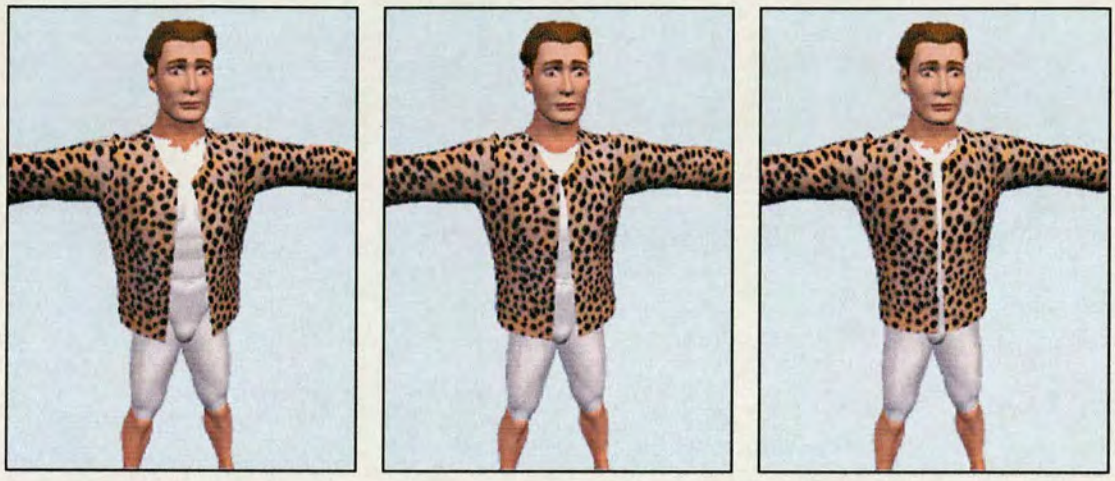


Figure 4.18: Final frames for scene *A* (octree, radial depth, capsule)

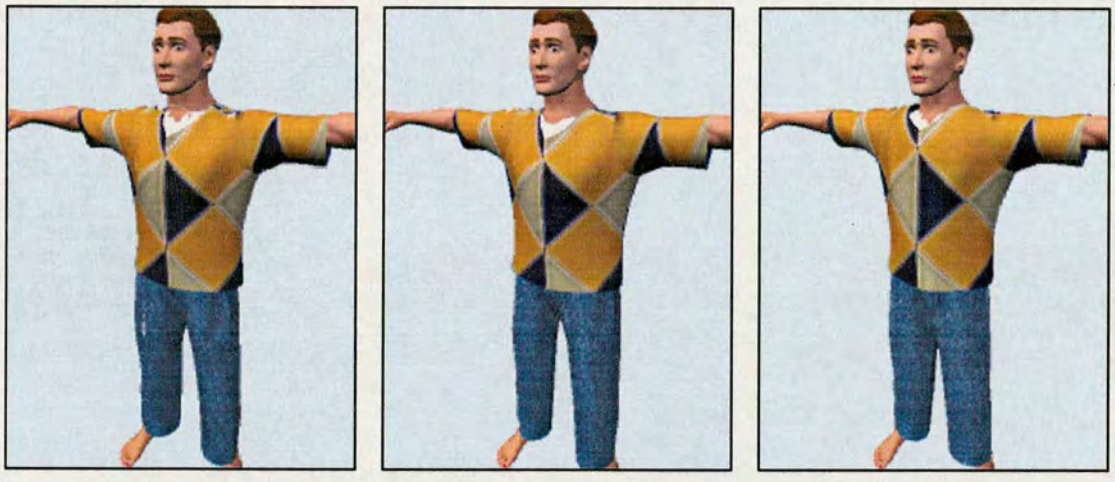


Figure 4.19: Final frames for scene *B* (octree, radial depth, capsule)

Figure 4.20: Final frames for scene *C* (octree, radial depth, capsule)

## 4.9 Summary

This chapter has concluded the exposition of the second point of the FIGMENT scheme by describing a second form of collision volume approximation: the 'radial depth' method. The algorithms for obtaining a best-fitting representation of a mannequin's body, a combination of both analytic and genetic methods, were provided in detail, followed by the calculations required to implement collision detection and response using that representation. Finally, timing data, accuracy analysis and visual results were provided from example modelling scenes to demonstrate the speed gains typically obtainable and the corresponding level of error introduced, with respect to both an optimised polygon-to-polygon method and the capsule method described in the previous chapter.

# Chapter 5

# Progressive Meshes

## 5.1 Introduction

The foregoing chapters have detailed methods for substantially reducing the simulation time for modelling clothing by using *(1)* a simplified physical model and *(2)* collision approximation techniques. A further increase in speed can be obtained by reducing the complexity (*i.e.* polygon count) of the clothing item models used in each simulation. A decrease in the polygon count of a cloth mesh means that less computations are required in determining the internal forces within the mesh as well as for collision detection and response. However, this decrease in complexity leads to a corresponding decrease in visual fidelity—the surface of the cloth is less able to follow the surface of the mannequin body, to exhibit smooth curvature, and to fold and crease appropriately (Fig. 5.1).

Nevertheless, it is possible to maintain a perception of fidelity by varying the complexity of the cloth meshes at different stages in the simulation. A simplified mesh may be used during the initial modelling frames, where the predominant dynamic forces are those of gravity and collision response, with relatively little effect on the final state of the mesh as it comes to rest. Conversely, during the final frames, the forces predominantly affecting the geometry of the mesh are those of the internal fabric mechanics and thus a higher complexity mesh is required to allow the cloth to bend and fold appropriately.

This chapter therefore presents a method of progressively increasing the complexity (polygon count) of a cloth mesh during simulation, at variable rates, in

order to further reduce the time in which the results are obtained. A number of previously developed techniques for reducing the polygon count of a mesh are considered with respect to their suitability for the FIGMENT scheme and one approach in particular is singled out as a viable basis for implementation.

A modified version of this approach is used within the FIGMENT scheme and therefore the algorithms employed to decimate high-complexity cloth meshes and to reconstruct them during the modelling process are described in detail, including the additional considerations necessitated by the dynamic nature of the surfaces of the meshes.

Finally, the speed advantage gained by using this approach is assessed by means of a number of example simulations, and the corresponding cost with respect to the accuracy and fidelity of the results is also considered.

Figure 5.1: Mannequin clothed with low-complexity garment

## 5.2 Polygon reduction methods

A number of methods have been developed for reducing the complexity of a polygon mesh in addition to other techniques which may be extended for this purpose. This section provides a brief survey of the main approaches available, before assessing the suitability of each for implementation within the FIGMENT scheme.

A seminal mesh decimation algorithm was presented by Schroeder, Zarge and Lorensen (1992) which relied on the successive deletion of vertices from meshes and the re-triangulation of surrounding vertices. Vertices are selected for deletion according to two criteria: either the vertex falls within a minimum distance to an average plane (based on surrounding polygons) or within a minimum distance to a hypothetical edge (defined between two adjacent vertices). The mesh vertices are also classified according to five different categories determined by their connection to surrounding vertices and polygons. Only vertices falling within certain categories are appropriate for deletion as indicated by each criterion. Any particular vertex may only be deleted if the re-triangulation of the 'gap' left by a deleted vertex (and its associated polygons) will be successful. The triangulation algorithm itself proceeds through recursively 'splitting' a loop of vertices by defining a polygon edge between two non-neighbouring vertices—each 'split' it chosen to be the one that gives the minimum aspect ratio between the two resultant loops, thus ensuring an optimal triangulation for any set of vertices.

One decimation method which builds on the approach of Schroeder *et al* (1992) is that of Hoppe, DeRose, Duchamp, McDonald and Stuetzle (1993). The 'mesh optimization' process aims to reduce the complexity of triangles meshes by successively applying transformations to selected *edges* of the meshes (*i.e.* the sides of polygons defined between pairs of vertices). Three edge transformations are possible: 'edge collapse', 'edge split' and 'edge swap'. The first brings the two ends of the edge together as one vertex, the second adds a vertex between the two ends of the edge and splits the adjacent triangles accordingly, and the third exchanges the shared edge between two triangles for a new edge defined between the opposing vertices of the triangles. For each iteration of the decimation process, the edge and the edge transformation are chosen with the aim of minimizing an energy function $E(M)$, where $M$ corresponds to the resultant mesh after any particular edge transformation. The energy function consists the sum of three parts, $E_{dist}(M)$, $E_{rep}(M)$, and $E_{spring}(M)$; the first term essentially represents the deviation of the surface of $M$ from that of the original model, the second term aims to penalize meshes with larger numbers of vertices, and the third is a regularizing term which

helps guide the optimization to a desirable local minimum. The edge to transform is chosen randomly from a set of candidate edges which consists of all edges that may lead to a beneficial transformation; the set initially contains all mesh edges and is updated after each transformation. If the selected edge may be 'legally' transformed (considering potential 'collapse', 'swap', and 'split' transformations in that order) then the transformation is performed, otherwise another randomly chosen edge is considered and so forth.

The method of Hoppe *et al* (1993) is further developed in Hoppe (1996) in the form of 'progressive meshes'. In this paper, the decimation algorithm has been altered in a number of ways. Firstly, the author determined that only one edge transformation, 'edge collapse', was required for an optimum polygon reduction; hence the 'edge split' and 'edge swap' transformations are no longer considered. This one transformation leads to the concept of 'progressive meshes'. An initial mesh $\vec{M} = M^n$ is subjected to a series of $n$ optimal edge collapse transformations, resulting in a base mesh $M^0$ of reduced complexity. The level-of-detail of this base mesh can then be progressively increased by performing a series of inverse transformations—'vertex splits'—until the original mesh $\vec{M}$ is restored. The authors suggest a number of applications for progressive meshes, including "smooth geomorphing of level-of-detail approximations, progressive transmission, mesh compression, and selective refinement."

The second modification to the decimation algorithm is that of the energy function $E(M)$ used to determine the optimum decimation path. Rather than choosing edges randomly from a candidate set, for each iteration of the algorithm the edge collapsing transformation is chosen which maximizes its estimated energy reduction $\Delta E$. As a consequence, the need for the $E_{rep}$ term in the original function is eliminated. In addition, two new terms are included in the energy function: $E_{scalar}(M)$ aims to preserve the scalar attributes of the mesh (*e.g.* surface colouring) and $E_{disc}(M)$ aims to preserve the discontinuity curves of the mesh (*e.g.* material boundaries, creases and shadow boundaries). Thus, the decimation process aims to

preserve, as closely as possible, both the shape and the appearance of the original mesh.

An alternative approach to mesh polygon reduction is offered via the technique of multiresolution analysis by Eck, DeRose, Duchamp, Hoppe, Lounsbery and Stuetzle (1995). Multiresolution analysis is a method of wavelet compression for meshes which possess subdivision connectivity, *i.e.* meshes obtained from a simplified base mesh by recursive 4-to-1 splitting of triangular faces. Progressively more complex meshes can thus be generated from the base mesh by introducing the wavelet coefficients required to reconstruct each set of four triangles from the previous one. Clearly, the necessity for subdivision connectivity places a considerable restriction on the use of multiresolution analysis; however, Eck *et al* detail an effective method for converting arbitrary meshes to multiresolution form and thus allowing an MRA simplification. To summarise, this is done by partitioning the original mesh $M$ into triangular regions and then constructing a base mesh $M^0$ with triangular faces corresponding to those regions. Recursive 4-to-1 splitting of faces is applied until a mesh $M^J$ is obtained which approximates $M$, contains the same order of faces, and exhibits subdivision connectivity. The method of Lounsbery (1994) is then used to obtain an MRA representation of $M^J$.

A further method of mesh simplification by means of 'simplification envelopes' is presented by Cohen, Varshney, Manocha, Turk, Weber, Agarwal, Brooks and Wright (1996). A simplification envelope for a polygonal mesh object $I$ is "a polygonal surface that lies within a distance of $\varepsilon$ from every point $p$ on $I$ in the same [or opposite] direction as the normal to $I$ at $p$." The simplification of the original mesh proceeds by defining two envelopes (outer and inner) each with a specified distance $\varepsilon$. One of two available algorithms then proceeds to reduce the complexity of the mesh by successive *hole creation* and *hole filling* stages. In each case, the algorithm is constrained to keep resultant meshes within the space defined by the two simplification envelopes. The two algorithms are *(1)* a 'local' algorithm which creates holes by attempting to remove a vertex from the mesh, and *(2)* a 'global' algorithm which attempts to create maximally-sizes holes. Clearly, the

results of using either algorithm will be quite different and each has a preferred application to varying sizes and complexities of models. The primary advantage of the method presented in Cohen *et al* (1996) is that it guarantees that all points of the simplified mesh are within a user-specifiable distance $\varepsilon$ from the surface of the original mesh and that all points of the original mesh are within a distance $\varepsilon$ from the surface of the simplified mesh.

Turk (1992) presents a technique for re-tiling polygon surfaces by allocating a set of surface points to a mesh, with an associated density indicating an estimate of the local curvature, and re-triangulates the mesh surface according to those points. The parameters of the triangulation algorithm may be adjusted to obtain a resultant mesh with the desired (lower) number of polygons.

Krishnamurthy and Levoy (1996) detail an algorithm for fitting smooth parametric surfaces (in particular, tensor product B-splines) to dense and irregular polygon meshes. This process may be used for polygon reduction by applying the algorithm to the mesh in question and subsequently re-triangulating (at a specified level of discreteness) the resultant parametric surfaces.

Finally, to conclude this survey of mesh simplification methods, Kalvin and Taylor (1996) present the 'Superfaces' algorithm for polygon reduction. The algorithm attempts to optimally partition the polygons in an arbitrary mesh into 'surface patches' (sets of connected polygons) where each surface patch corresponds to a 'superface' (a non-planar polygon). The mesh is then approximated, at a reduced complexity, by triangulating the set of corresponding superfaces. Permitting the approximation of the original mesh to within a user-specified tolerance, the Superfaces algorithm is "very efficient [and] a practical method for simplifying very large meshes, such as those derived from medical CT and MRI data."

## 5.3 Assessment of polygon reduction methods

The main factor to take into account when assessing the suitability of the above mesh simplification method for implementation is that of *topology*. If the approximated mesh retains neither local nor global topology with respect to the

original then the problems in implementing a smooth transformation from the former to the latter during cloth simulation are likely to be insurmountable; particularly so since the vertices of the mesh will be constantly changing during the simulation, and thus any increase in complexity (*i.e.* in vertex number) must involve the vertex positions of the transformed mesh being computed from the previous mesh (rather than simply specifying absolute positions). This dynamic aspect of the mesh will most certainly limit the feasibility of implementing the majority of available techniques.

Furthermore, a method is preferable if it allows a large number of intermediate steps between the initial simplified mesh and the final complex mesh. These intermediates should only involve local topological modifications in the mesh, the reason being that any major changes in the topology of the mesh would introduce problems with respect to the physical model of the cloth itself. Such changes could introduce fatal instabilities into the dynamic system and would require considerable recalculation of other information defining the state of the cloth mesh, including:

- node positions and velocities
- joints between adjacent sections and other cloth meshes
- equilibrium ('at rest') states of individual sections
- appearance attributes, *e.g.* texture mapping information

With these considerations in mind, the approaches offered by Turk (1992), Krishnamurthy and Levoy (1996) and Kalvin and Taylor (1996), which do not preserve global topology, are immediately seen to be unsuitable. Eck *et al* (1995) maintains global topology between successively simplified meshes, yet the change between intermediates (a four-fold increase in complexity) means that although its implementation is possible in theory, other alternatives are preferable.

The 'local' simplification algorithm of Cohen *et al* (1996) offers a viable option, since it can produce a large number of intermediate meshes with only local topological changes. However, the implementation of the initial algorithm for determining simplification envelopes and the subsequent code required to determine the validity of a potential 'hole creation' are far from straightforward. Furthermore,

the prescribed algorithm for 'hole filling', although efficient in terms of pure simplification, does not preserve local topology as well as Hoppe (1996).

Hoppe (1996) presents a highly appropriate approach to cloth mesh simplification and progressive reconstruction. The decimation algorithm is straightforward to implement, with potential for modification according to the specific constraints of the FIGMENT application. In addition, the transformation between intermediates (a 'vertex split' operation) is simple to perform, offers a minimal change in local topology, and does not require a substantial adjustment in the physical state of the cloth mesh. However, the algorithm as originally presented requires some development if it is to be used for the reconstruction of dynamic meshes which possess physical properties in addition to geometrical and visible attributes. This development is the concern of the following section.

## 5.4 A modified progressive mesh method

A number of modifications need to be made to the progressive mesh algorithms of Hoppe (1996) for implementation within the FIGMENT scheme. Firstly, the decimation algorithm may result in simplified meshes which contain an uneven distribution in the sizes of its sections, *e.g.* large sections in flatter areas and smaller sections in areas of high curvature or at corners. This can result in poor physical modelling due to the effect of bending forces which occur at the joints between sections; in addition, the larger sections can cause excessive deformation of neighbouring smaller sections, leading to instability problems. The algorithm must therefore be modified in order to maintain the homogeneity of the mesh.

Secondly, the dynamic nature of the mesh must be taken into account. The reconstruction of progressive meshes in Hoppe (1996) is implemented for static models in which the positions of its vertices do not change (with respect to their local coordinate system) during the process, except where vertices are split. During cloth modelling, however, the mesh is being continually deformed such that specifying an *absolute* position for created vertices (and their affected neighbours) is insufficient.

These positions must, instead, be estimated relative to the current geometry of the entire mesh.

Finally, the additional information required to perform physical modelling, beyond the mere geometry of the mesh, must be correctly handled by the decimation and reconstruction algorithms. In order to compute the internal forces within the mesh at any particular point, the current deformation of the mesh sections must be considered with respect to the equilibrium state of the mesh. In addition, the 'physical' attributes of individual nodes, sections and joints need to be correctly updated after each modification of the mesh (*i.e.* a vertex-split transformation).

The following three sections detail the progressive mesh algorithms as developed within the FIGMENT scheme.

## 5.5 The decimation algorithm

The decimation algorithm is based on an energy minimization principle as with the original method presented by Hoppe *et al* (1993). The algorithm performs $n$ successive edge-collapse transformations, whereby a pair of connected mesh vertices is combined into a single one (Fig. 5.2), on an arbitrary mesh $\tilde{M} = M^n$ of triangular faces until a suitably simplified mesh $M^0$ has been obtained. The particular edge-collapse transformation to perform at each stage for an intermediate mesh $M^i$ is chosen to be the one which minimises an energy function $E(M)$ which is evaluated for each possible resultant mesh $M^{i-1}$. The original mesh $\tilde{M}$ may subsequently be obtained from the simplified mesh $M^0$ by a series of $n$ reverse (vertex-split) transformations. The key to a successful decimation result is to use a suitable energy function which will aim to retain the primary characteristics of the original mesh.

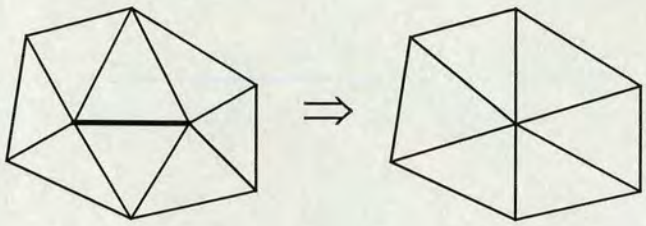


Figure 5.2: Edge-collapse transformation

The energy function

The energy function used for the FIGMENT decimation algorithm is defined as follows:

$$E(M) = E_{dist}(M) + E_{perim}(M) + E_{area}(M) \tag{5.1}$$

This function differs from that used by Hoppe (1996) in several ways. The first term substantially corresponds to the $E_{dist}$ term in the latter. The $E_{spring}$, $E_{scalar}$ and $E_{disc}$ terms have not been included, however, for the following reasons.

The $E_{spring}$ term was intended originally as a regularizing term, "most important in the early stages of the optimization," because it ensured that a local minimum was always present during the optimization process. Without this term, it was found that 'spikes' could appear on the surface of a mesh $M$ when adjusting the positions of its vertices in order to determine the minimum value for the other terms of $E(M)$ (the reader is referred to Hoppe *et al* (1993) for the details of this problem). However, the FIGMENT decimation algorithm does not perform vertex position adjustment when each potential edge-collapse transformation is considered for suitability and thus the regularizing $E_{spring}$ term is redundant. Vertex position adjustment is not performed simply because the dynamic, deformed nature of the cloth meshes during physical simulation would render this optimization step practically worthless and next to impossible to implement efficiently; the adjusted positions of mesh nodes after a vertex-split transformation would need to be estimated in the same way as for the newly created node (see Section 5.7 for the details of the calculations required).

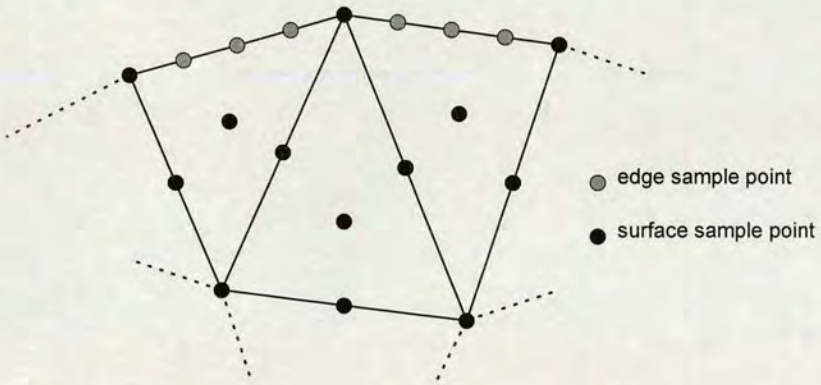



Figure 5.3: Sampling of surface points and edge points from model

The $E_{scalar}$ and $E_{disc}$ terms of Hoppe (1996) are also discarded since they apply mainly to models which rely on certain appearance attributes being preserved, such as vertex colour $(r,g,b)$ components, and 'crease' and 'shadow' effects. The appearance of the clothing models used by the FIGMENT schemes are generally defined by geometry and texture mapping. The nature of texture mapping is such that a change in the underlying geometry can be straightforwardly translated to a change in texture coordinates in a way that results in little visual difference between one geometrical change and another. In other words, there is no particular aspect of the texture mapping effect that is important to preserve during simplification beyond the correspondence of two-dimensional texture mapping coordinates to three-dimensional vertex coordinates. The contribution of the $E_{scalar}$ and $E_{disc}$ terms are therefore not significant enough to merit inclusion in the energy function of the FIGMENT decimation algorithm.

As mentioned above, the $E_{dist}(M)$ function corresponds here to that of Hoppe (1996) by providing a measure of the extent to which the surface of the mesh $M$ follows that of the original mesh $\vec{M}$. The decimation algorithm initially defines a set of sample points $X = \{\mathbf{x}_1, \mathrm{K}\ , \mathbf{x}_n\}$ from the surface of $\vec{M}$. These points are taken from the vertices of the mesh, the centre points of its edges, and the centre points of its faces (Fig. 5.3). No sample points are taken from unique edges, *i.e.* the perimeter of the mesh, but a second set $X' = \{\mathbf{x}'_1, \mathrm{K}\ , \mathbf{x}'_m\}$ is defined to contain a specified number of equally-spaced sample points from each unique edge. This second set of

sample points is used within another term of the energy function (see below) to prevent the perimeter edges of the mesh from becoming unacceptably distorted. The $E_{dist}$ term is calculated to be proportional to the sum of the squared minimum distances of each point $\mathbf{x}_i$ of $X$ from the surface of $M$:

$$E_{dist}(M) = k\sum_{i=1}^{n} d_{surf}^2(\mathbf{x}_i, M) \tag{5.2}$$

Similarly, the energy term $E_{perim}$ is intended to represent the deviation of the perimeter (unique) edges of mesh $M$ from that of the original mesh and is calculated to be proportional to the sum of the squared minimum distances of each point $\mathbf{x}_i'$ of $X'$ from the surface of $M$:

$$E_{perim}(M) = k'\sum_{i=1}^{m} d_{perim}^2(\mathbf{x}_i', M) \tag{5.3}$$

The third and final term of the energy function, $E_{area}$, is required in order to maintain a relatively homogenous mesh surface with respect to the areas of the triangular faces as the decimation algorithm proceeds. As indicated previously, the decimation of a typical cloth mesh would result in large flat areas consisting of a few large sections in contrast to a high number of small sections remaining at points of high curvature or creasing. Such a mesh would be less suitable for the purposes of even and realistic cloth modelling and hence the $E_{area}$ term has been introduced to counteract those results. $E_{area}$ is calculated to be proportional to the average area of the faces eliminated from the mesh by a potential edge collapsing transformation. Hence, if $F = \{\mathbf{f}_1, \mathrm{K}, \mathbf{f}_n\}$ is the set of sections removed by the transformation to mesh $M$:

$$E_{area}(M) = k_{area}\frac{1}{n}\sum_{i=1}^{n} A(\mathbf{f}_i) \tag{5.4}$$

Evaluating the energy function

The first two terms of the energy function, $E_{dist}$ and $E_{perim}$, require the calculation of the minimum distance of sample point from the surface and perimeter edges of a mesh $M_i$, respectively. The robust method of determining the minimum distance to the surface of a mesh is to compute the shortest distance from the point to each face of the mesh and take the lowest of these values. This is computationally inefficient, however, and so a heuristic method is used instead whereby the point is only compared to one face **f** and its immediate neighbours, where **f** was the closest face of the previous mesh $M_{i+1}$ to that point. If the face **f** is actually eliminated by the edge-collapse transformation under consideration, then only its remaining neighbours are used in the minimum distance calculation. Thus, at the end of each decimation step, every sample point of $X$ is associated with a closest face of the resultant mesh $M_i$.

The procedure for calculating the minimum distance of each sample point of $X'$ to the perimeter edges of $M_i$ uses the same heuristic algorithm. In practice, the sample point is compared against *every* edge of the face **f** and its immediate neighbours, rather than only the unique edges of the mesh. This is done in order to simplify the implementation, since the required code to determine the distance of a point to the surface of a face and the distance of a point to the edges of a face is practically identical, except for cases whether the nearest surface point lies 'within' the edges of the face. Although this approach could potentially lead to the 'slippage' of the perimeter edges of the simplified mesh from that of the original mesh, if the number of perimeter edge sample points and the value of $k'$ in equation (5.3) are sufficiently high, then this problem is easily avoided.

Selecting the edge-collapse transformation

Every edge of a mesh is considered for an edge-collapse transformation. Since the transformation consists of merging the two vertices of the edge into one, an

infinite number of potential transformations are possible. However, as with Hoppe (1996), only three possibilities are considered which together provide a satisfactory range of transformations to consider. The three possible positions for the single vertex are: *(1)* that of one end of the edge, *(2)* that of the other end, or *(3)* that of the midpoint of the edge. The energy function for each of the three potential resultant meshes is computed and the overall transformation which minimizes the energy function is chosen as that included in the decimation path.

The selected edge-collapse transformation at each stage is simply recorded as the two indices of the edge vertices $\mathbf{v}_i$ and $\mathbf{v}_j$ along with a scalar value $\alpha \in \{0,0.5,1\}$ indicating the relative position of the resultant vertex $\mathbf{v}_i'$ such that:

$$\mathbf{v}_i' = (1-\alpha)\mathbf{v}_i + \alpha\mathbf{v}_j \tag{5.5}$$

Thus the decimation algorithm outputs a series of $n$ edge-collapse specifications, describing the optimal decimation path, plus a simplified base mesh $M^0$.

Texture mapping coordinates

The clothing meshes used within FIGMENT applications generally contain texture mapping information in the form of a two-dimensional coordinate (corresponding to a point on a texture image) associated with each three-dimensional vertex of the mesh. For each single vertex resulting from an edge-collapse transformation, its associated texture mapping coordinate $\mathbf{u}_i'$ is simply estimated from the coordinates, $\mathbf{u}_i$ and $\mathbf{u}_j$, of the two edge vertices as follows:

$$\mathbf{u}_i' = (1-\alpha)\mathbf{u}_i + \alpha\mathbf{u}_j \tag{5.6}$$

## 5.6 Seamed meshes

An additional complication to the decimation process is presented by those clothing item models which consist of multiple meshes, held together during physical

simulation by 'virtual seams' (see Section 2.4). The fact that nodes are seamed between meshes effectively means that edges are shared between those meshes; if a seamed edge is collapsed in one mesh, then its counterpart must also be collapsed in such a way as to avoid disrupting the seam as far as possible.

The FIGMENT scheme deals with multiple mesh clothing models by considering them to be one seamless mesh when evaluating the energy function during the decimation process. If a seamed edge (*i.e.* one in which both vertices are seamed) is considered for collapse, then its counterpart is also considered to be collapsed in the mesh to which the energy function is applied. If such an edge is selected for collapse, then one of the seam connections between the two meshes is lost in the decimation process, although the other one remains between the two resultant vertices (Fig. 5.4).

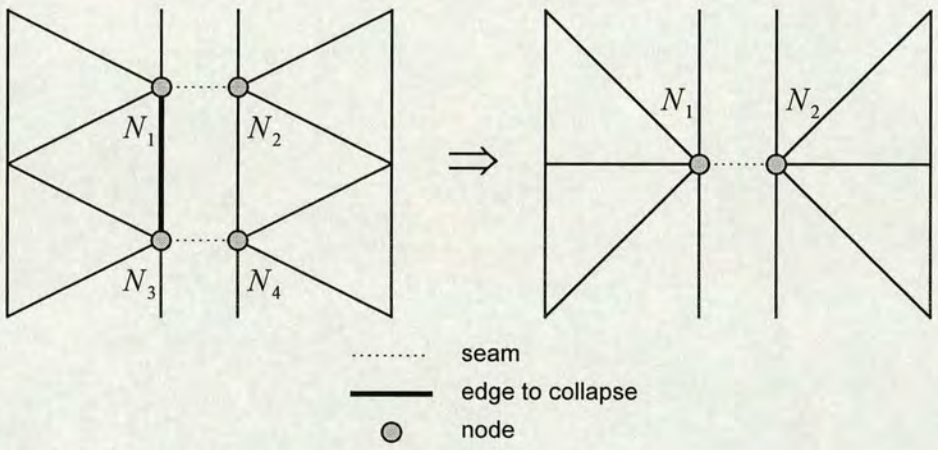



Figure 5.4: Collapse of an edge seamed at both ends

Furthermore, if an edge with only one seamed vertex is considered for collapse, then the decimation algorithm ensures that the other vertex is the one removed by the edge-collapse transformation; in this way, the seam connection is retained (Fig. 5.5).

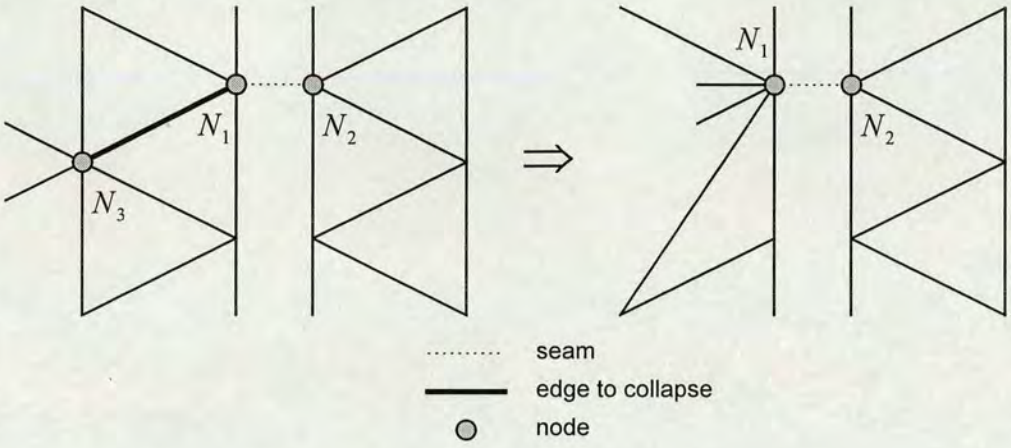



Figure 5.5: Collapse of an edge seamed at only one end

This complication added by the presence of seams between meshes means that the output from the decimation algorithm must contain additional information to allow the correct reconstruction of the simplified model. Firstly, each edge-collapse specification must include an index value indicating to which mesh of the model the transformation is to apply.[1] Secondly, the collapse of a seamed edge requires the immediate collapse of its counterpart; there should be some indication in the output, then, as to whether the edge-collapse transformation currently specified necessitates the transformation which immediately follows it.

It will be noted that the collapse of a pair of seamed edges is effectively equivalent to only one simplification transformation in the decimation process. Thus, if the decimation algorithm is required to produce a simplified base mesh $M^0$ by means of $n$ intermediate meshes and $m$ of the $n$ edges selected for collapse are, in reality, pairs of seamed edges, then the algorithm will output $n+m$ edge-collapse transformations as its optimal decimation path.

Although the approach described above is effective in many cases, its main disadvantage is the loss of seam connections through removed vertices, the effect of which becomes more serious for models which are greatly simplified. An alternative approach, therefore, is to altogether avoid collapsing edges which involve seamed

[1] This is required since the clothing meshes are considered to be distinct entities during physical simulation, with their nodes being individually indexed such that simply specifying an edge by the indices of its end nodes is ambiguous for a multi-mesh model.

vertices; either *(a)* those which possess at least one seamed vertex or *(b)* those in which both vertices are seamed. The former method is somewhat extreme and prevents the meshes from being simplified at all in the neighbouring region of the seamed edges. However, the latter case works well in practice, by maintaining the seams of the model without restricting the potential simplification to any great extent. For particularly complex (*i.e.* high polygon count) models, a combination of the above methods is often appropriate. If $n$ edge-collapse transformations are to be applied to such a model, the first $n_1$ operations may be performed such that seams are *not* preserved and then $n_2$ operations such that seams *are* preserved, where $n = n_1 + n_2$. In this way, the number of seams removed does not adversely affect the overall seaming of the garment and a more favourable homogeneity of mesh sections is obtained.

## 5.7 The reconstruction algorithm

The reconstruction process for the progressive meshes of Hoppe (1996) is straightforward. A series of vertex-split transformations, corresponding to each edge-collapse in the decimation path, are applied to the simplified base mesh $M^0$. The positions of the two vertices resulting from a vertex-split may be specified as absolute coordinates in the local system of the mesh. In addition, information must be supplied as to which of the polygons surrounding the original vertex should subsequently be defined by one, or the other, of the two resultant vertices.

Reconstructing a cloth mesh which is constantly undergoing dynamic deformation according to physical modelling is a significantly more complex task. Newly-created nodes cannot have their locations specified absolutely, but must instead be estimated with respect to the current geometry of the mesh. The current velocity vectors of created or modified nodes need to be adjusted appropriately, as do the texture coordinates of texture mapped meshes and the point masses of nodes. Furthermore, since the internal elastic forces acting on the nodes of the mesh are computed with respect to the equilibrium state of the mesh, the modelling system must always have access to information which defines this equilibrium state. The

information is most usefully represented as the 'at rest' dimensions (*i.e.* edge lengths) of every section in the mesh. Clearly, this data must change as the topology of the mesh changes with each vertex-split transformation.

Reconstruction scripts

In order to make this information available to the actual modelling system, the FIGMENT scheme incorporates a precomputation stage which outputs a 'reconstruction script' for every simplified model (consisting of seamed progressive meshes). The input to the precomputation software is the series of optimal edge-collapse transformations output by the decimation algorithm *and* the original clothing model itself. The reconstruction script specifies the series of vertex-split transformations required to restore the original complexity of the model along with the additional information (listed in Section 5.3) which is necessary to maintain the geometry, appearance, and physical attributes of the mesh.

Before detailing the form of the script, the following explanation of the internal representation of a FIGMENT cloth mesh is required. As described in Chapter 2, the mesh is defined as a set of *nodes*, a set of *sections* (defined as an ordered triad of nodes) and a set of *joints* (defined as a pair of adjacent sections). A simplified mesh, however, possesses the *same* set of nodes as the original mesh from which it was obtained. The reason for this is purely for ease of implementation. Nodes are referenced according to an index value; by beginning with the final set of nodes, the indices need not be changed as nodes are 'created' by vertex-split transformations. In addition, all memory allocation is performed prior to the modelling process. A Boolean value is associated with each node to indicate whether it is referenced within the current mesh; a 'created' node merely needs to be 'referenced' after setting its other attributes appropriately (*e.g.* position, velocity, mass, *etc.*).

The reconstruction script consists of a sequence of vertex-split definitions with the format shown in Fig. 5.6, where `<node>` and `<section>` correspond to the identifying indices of each component within the mesh and `<vector>` corresponds to three floating-point values.

```
SPLITVERTEX     <node_to_split> <node_to_create>
TEXTURECOORDS   <texture_u1> <texture_v1> <texture_u2> <texture_v2>
CREATESECTION   <section_to_create> <node1> <node2> <node3>
                <unstressed_edge_vector1> <unstressed_edge_vector2>
CREATESECTION   ...
...
UPDATESECTION   <section_to_update> <reattachment_flag>
                <unstressed_edge_vector1> <unstressed_edge_vector2>
UPDATESECTION   ...
...
FROMSECTION     <section> <coefficient_vector1> <coefficient_vector2>
FROMSECTION     ...
...
```

Figure 5.6: Reconstruction script format

The SPLITVERTEX field indicates the index of the node to be 'split' and the index of the node to 'create'; in practice, the latter already exists and merely needs to be referenced. The TEXTURECOORDS field, if present, specifies the updated texture coordinates to be associated with each node. These values are simply taken from the original and intermediate meshes during the decimation process.

The CREATESECTION field specifies a new section which must be created by this vertex-split transformation. In addition to specifying the three nodes of the section, two of which will be the 'split' and 'created' nodes, this field must also provide information regarding the equilibrium state of the section. This could be done by merely specifying the 'at rest' dimensions (*i.e.* edge lengths) of the section. However, the FIGMENT scheme allows arbitrary geometric transformations to be applied to cloth meshes prior to modelling and therefore the CREATESECTION field provides two three-dimensional vectors corresponding to two unstressed edges of the section as it occurs in the original or intermediate mesh; by applying the same geometric transformation to these vectors, the correct equilibrium dimensions can be easily computed. There should be as many CREATESECTION fields as there are sections removed by the reverse edge-collapse transformation during simplification; there ought always to be at least one.

In a similar way, the UPDATESECTION field specifies the updated equilibrium state for those neighbouring sections affected by the vertex-split transformation. An additional Boolean flag is included to indicate whether the updated section should remain 'attached' to the 'split' node or should be 'reattached' to the 'created' node (Fig. 5.7). There should be as many UPDATESECTION fields as there are sections surrounding the 'split' node prior to the transformation.

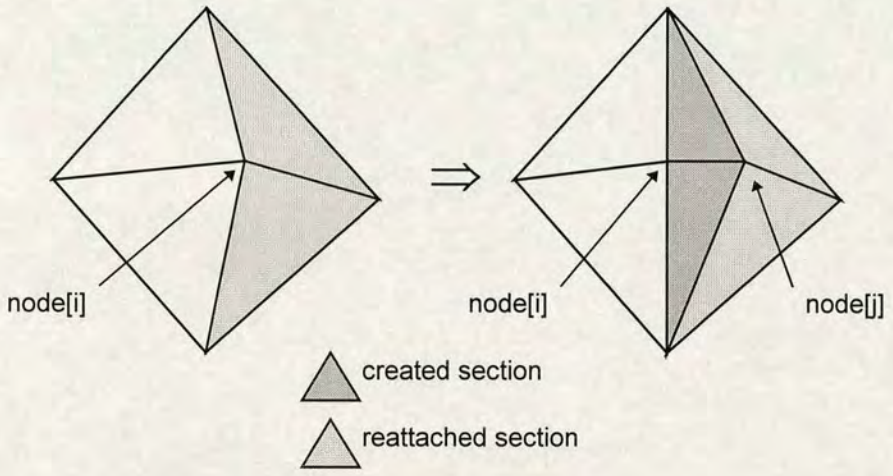



Figure 5.7: Reattachment of sections to newly-created node

Finally, the FROMSECTION field allows the positions of the two resultant nodes to be estimated from the positions of the surrounding sections prior to the transformation. As explained previously, absolute coordinates cannot be specified for the nodes due to the deformed state of the mesh during physical modelling. However, if a position can be estimated relative to an individual section, then despite the global change in location of the section and the local change due to the effects of physical modelling, a suitable initial position for the node can be calculated as the mean estimated position from neighbouring sections. Any error in the estimated position will quickly be corrected by the elastic properties of the mesh.

The crucial question is therefore how to best estimate a node's position relative to a section. Geometrically, a point in three-dimensional space may be represented as the weighted sum of three position vectors (in the normative case, three unit vectors corresponding to the three primary axes):

$$\overline{u} = w_1 \cdot \overline{v}_1 + w_2 \cdot \overline{v}_2 + w_3 \cdot \overline{v}_3 \tag{5.7}$$

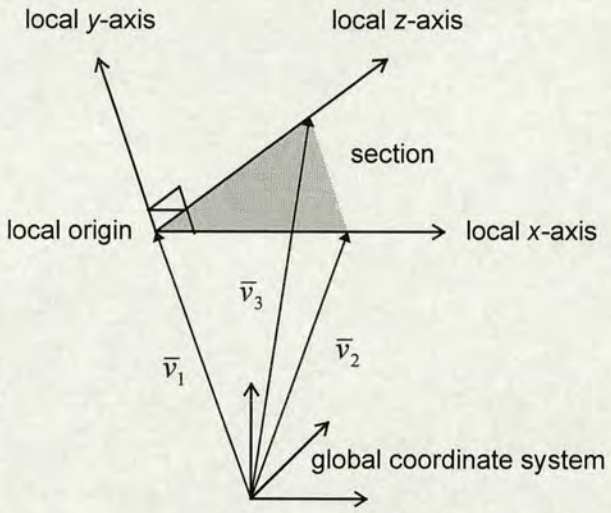
It would therefore be theoretically possible to express the position vector of a node as the weighted sum of the position vectors of the three nodes of the section. This could allow a suitable estimate of the node's position relative to that section within a deformed mesh. However, this method becomes less reliable in cases where the *origin* of the position vectors lies near the *plane* of the section, since the weightings become large values and thus any error introduced can be greatly magnified. In fact, in cases where the origin lies directly in the plane of the section, calculation of the weightings is impossible since the set of simultaneous equations represented has no solution.

A better approach is to use three position vectors with a *local* origin, chosen in order to minimize the coplanarity problem. The solution taken here is to use the first node of a section as the local origin, and two sides of the section plus its normal vector as the three position vectors (Fig. 5.8). Thus, if $\overline{v}_1$, $\overline{v}_2$ and $\overline{v}_3$ are the position vectors of the three nodes of the section, $\overline{n}$ is its normal vector, and $\left(w_{a1}, w_{a2}, w_{a3}\right)$ and $\left(w_{b1}, w_{b2}, w_{b3}\right)$ are the two coefficient vectors specified by the `FROMSECTION` field, then estimates of $\overline{v}_a$ and $\overline{v}_b$ (the position vectors of the 'split' and 'created' nodes) may be calculated as follows:

$$\overline{v}_a = \overline{v}_1 + w_{a1} \cdot \overline{n} + w_{a2} \cdot \left(\overline{v}_2 - \overline{v}_1\right) + w_{a3} \cdot \left(\overline{v}_3 - \overline{v}_1\right) \tag{5.8a}$$

$$\overline{v}_b = \overline{v}_1 + w_{b1} \cdot \overline{n} + w_{b2} \cdot \left(\overline{v}_2 - \overline{v}_1\right) + w_{b3} \cdot \left(\overline{v}_3 - \overline{v}_1\right) \tag{5.8b}$$

As indicated above, estimates of the two node positions are computed in this way from each section surrounding the 'split' node and the final positions taken to be the mean positions. This method of estimation has proved to be most effective in practice; the level of temporary distortion in the mesh due to poor estimations of node positions is almost imperceptible.




Node positions are estimated with respect to the local (non-orthogonal) coordinate system of each neighbouring section. This estimation can therefore be expressed in terms of the three position vectors of the corner nodes of each section.

Figure 5.8: Local coordinate system for estimation of node positions

The reconstruction scripts therefore provide sufficient information for cloth meshes, undergoing dynamic deformation, to be effectively reconstructed from their simplified versions during modelling simulations. The actual rate at which reconstruction occurs (*i.e.* at which vertex-split transformations are applied) may be varied, of course; a suitable rate is one which balances the speed advantage of lower rates (in which less complex intermediate meshes are present for longer) and the accuracy of higher rates (in which more 'flexible' meshes appear sooner). The speed and accuracy of various rates of progression are examined in the following two sections.

It should be noted in passing that the above script format (Fig. 5.6) is highly space inefficient and is only presented thus for the purposes of readability. In

practice, an implementation of the FIGMENT scheme should use a compressed, tokenized format to minimize download and parsing times.

Reconstruction algorithm

A number of steps are required in order to perform one vertex-split transformation on a progressive cloth mesh during modelling. The algorithm used in the FIGMENT scheme proceeds as follows:

1. The joints (see Section 2.3) within the mesh affected by the vertex-split transformation are removed, *i.e.* those occurring between the sections which surround the node to be split. *Note*: it is the internal representation of the joints, by which the physical bending forces are computed, which are discarded—not the actual geometry of the mesh.
2. The mass contribution of the surrounding sections are removed from the nodes between which they are defined. It will be recalled that the mass of a section, determined by its equilibrium area, thickness, and density, is equally distributed to its three nodes. This step in the reconstruction algorithm simply subtracts this mass contribution.
3. The 'new' node is 'created' as specified by the `SPLITVERTEX` field. It will be recalled that the FIGMENT scheme in practice simply changes the 'referenced' flag of the node.
4. The new sections are created as specified by the `CREATESECTION` fields. Each pair of specified vectors, corresponding to the sections sides in the original mesh, are geometrically transformed (if required) and the equilibrium dimensions of the corresponding section are calculated.
5. The positions of the 'split' and 'created' nodes are estimated from the sections surrounding the 'split' node as specified by the `FROMSECTION` fields. The velocity of the 'created' node is initially set to be that of the 'split' node. The texture coordinates of the vertices of the mesh model, corresponding to the nodes, are updated according to the `TEXTURECOORDS` field.

6. The equilibrium dimensions of those sections affected by the vertex-split (*i.e.* those surrounding the 'split' node prior to the transformation) are updated as specified by the `UPDATESECTION` fields. The 'split' node is replaced by the 'created' node in those sections indicated by the reattachment flag of the fields.
7. The mass contributions of all affected sections (those surrounding the 'split' node, plus those created by the transformation) are added to the nodes between which they are defined. The total mass contribution added should equal that removed in step 2 if the reconstruction algorithm is correctly implemented.
8. New joints are created between adjacent pairs of affected (including new) sections.

It will be observed that the steps of the reconstruction algorithm need not necessarily be performed in the above sequence.

## 5.8 Speed comparisons

In order to assess the speed increases afforded by using progressive meshes for modelling clothing, two typical modelling scenes were simulated on three different platforms—a 200MHz Pentium PC, a 170MHz Sun ULTRASparc and a 180MHz MIPS R5000 Silicon Graphics O2—using four different combinations of mesh progression.

The first four simulations consisted of a male mannequin being clothed with a sweater (originally with 1723 cloth sections to which 800 edge-collapses were applied resulting in 177 sections); the second four consisted of a female mannequin being clothed with a dress (originally with 1103 sections to which 500 edge-collapses were applied resulting in 141 sections). Table 5.1 provides the details of the eight simulations run. In every case, the simulation was performed for 6.0 'virtual' seconds, with a time-step of 0.001 'virtual' seconds, standard internal force computation methods (see Section 2.3) and radial depth collision approximation.

| Simulation | Description | Number of sections at start of simulation | Number of sections at end of simulation | Number of iterations between vertex-split operations | Number of vertex-splits applied beforehand | Number of vertex-splits applied during simulation |
|---|---|---|---|---|---|---|
| *1A* | Male with sweater | 1723 | 1723 | — | 800 | 0 |
| *1B* | Male with sweater | 1331 | 1723 | 20 | 600 | 200 |
| *1C* | Male with sweater | 937 | 1723 | 10 | 400 | 400 |
| *1D* | Male with sweater | 177 | 1723 | 5 | 0 | 800 |
| *2A* | Female with dress | 1103 | 1103 | — | 500 | 0 |
| *2B* | Female with dress | 712 | 1103 | 20 | 300 | 200 |
| *2C* | Female with dress | 321 | 1103 | 10 | 100 | 400 |
| *2D* | Female with dress | 321 | 1103 | 5 | 100 | 400 |

Table 5.1: Details of example simulations

Tables 5.2, 5.3 and 5.4 show the timing data obtained for each simulation scene on each hardware platform: the time required for one iteration of computation both at the beginning and end of mesh progression, the overall time for the simulation, and the speed gain with respect to the equivalent non-progressive case.

| Simulation | Time for iteration at start of progression | Time for iteration at end of progression | Total duration of simulation | Speed gain with respect to non-progressive case |
|---|---|---|---|---|
| *1A* | 61.1 ms | 62.8 ms | 253 s | — |
| *1B* | 46.9 ms | 62.9 ms | 227 s | 11 % |
| *1C* | 33.8 ms | 62.9 ms | 200 s | 27 % |
| *1D* | 8.61 ms | 62.9 ms | 148 s | 71 % |
| *2A* | 34.7 ms | 35.3 ms | 144 s | — |
| *2B* | 23.6 ms | 35.2 ms | 124 s | 16 % |
| *2C* | 11.4 ms | 35.2 ms | 100 s | 44 % |
| *2D* | 11.4 ms | 35.2 ms | 123 s | 17 % |

Table 5.2: Timing results for progressive mesh simulations (Pentium PC)

| Simulation | Time for iteration at start of progression | Time for iteration at end of progression | Total duration of simulation | Speed gain with respect to non-progressive case |
|---|---|---|---|---|
| *1A* | 125 ms | 132 ms | 520 s | — |
| *1B* | 96.7 ms | 132 ms | 469 s | 11 % |
| *1C* | 70.7 ms | 132 ms | 414 s | 26 % |
| *1D* | 21.2 ms | 132 ms | 311 s | 67 % |
| *2A* | 67.4 ms | 68.9 ms | 276 s | — |
| *2B* | 48.3 ms | 69.1 ms | 243 s | 14 % |
| *2C* | 24.8 ms | 70.1 ms | 199 s | 39 % |
| *2D* | 25.3 ms | 69.5 ms | 240 s | 15 % |

Table 5.3: Timing results for progressive mesh simulations (Sun UltraSPARC)

| Simulation | Time for iteration at start of progression | Time for iteration at end of progression | Total duration of simulation | Speed gain with respect to non-progressive case |
|---|---|---|---|---|
| *1A* | 122 ms | 124 ms | 492 s | — |
| *1B* | 92.5 ms | 123 ms | 436 s | 13 % |
| *1C* | 66.9 ms | 123 ms | 384 s | 28 % |
| *1D* | 19.1 ms | 125 ms | 286 s | 72 % |
| *2A* | 67.9 ms | 69.0 ms | 415 s | — |
| *2B* | 47.4 ms | 69.5 ms | 380 s | 9 % |
| *2C* | 23.5 ms | 69.4 ms | 334 s | 24 % |
| *2D* | 24.8 ms | 70.1 ms | 379 s | 9 % |

Table 5.4: Timing results for progressive mesh simulations (Silicon Graphics O2)

## 5.9 Accuracy comparisons

In this section, accuracy analysis[2] is provided for the simulations detailed in the previous section. Fig. 5.9(a) plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes in scenes *1B*, *1C* and *1D* compared with scene *1A*. Similarly, Fig. 5.9(b) plots the mean and standard deviation of the error between corresponding nodes of the cloth meshes in scene *2B*, *2C* and *2D* compared with scene *2A*. In both cases, the error (*i.e.* distance between corresponding nodes) is expressed as a percentage of the total height of the mannequin. During the period of simulation in which mesh reconstruction (progression) is occurring there is no one-to-one correspondence between the nodes of meshes in progressive and non-progressive simulations, and therefore the relative vertex error has been computed

[2] The reader is referred to Section 2.7 for the details of the accuracy analysis method used here.

only for the simulation frames subsequent to the point at which the meshes are fully reconstructed.

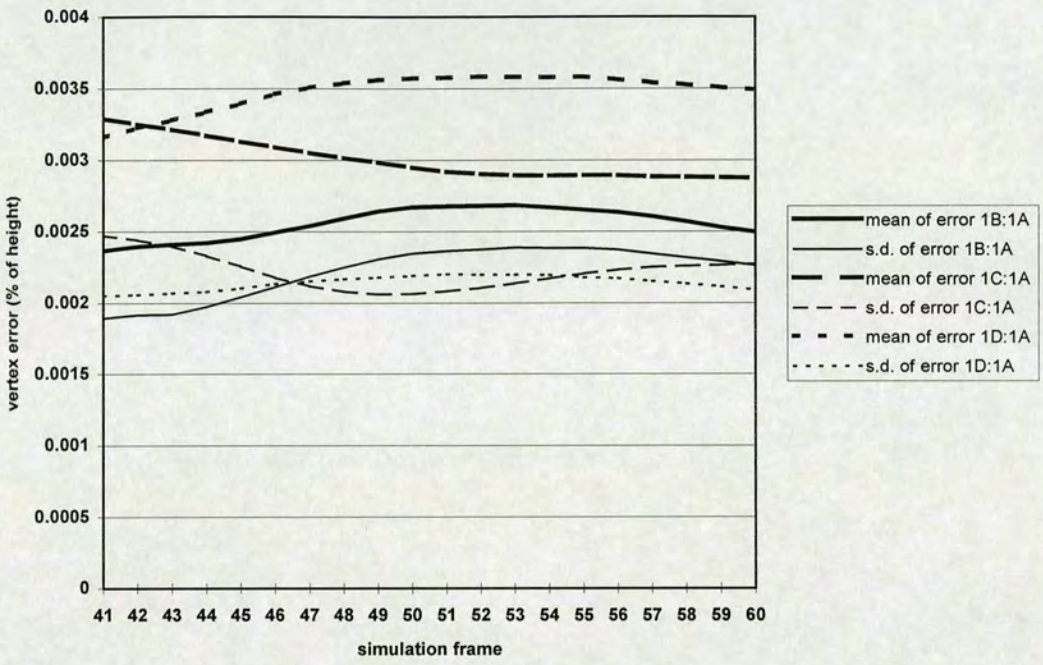



Figure 5.9(a): Accuracy comparison (scenes *1B*, *1C*, *1D* against scene *1A*)

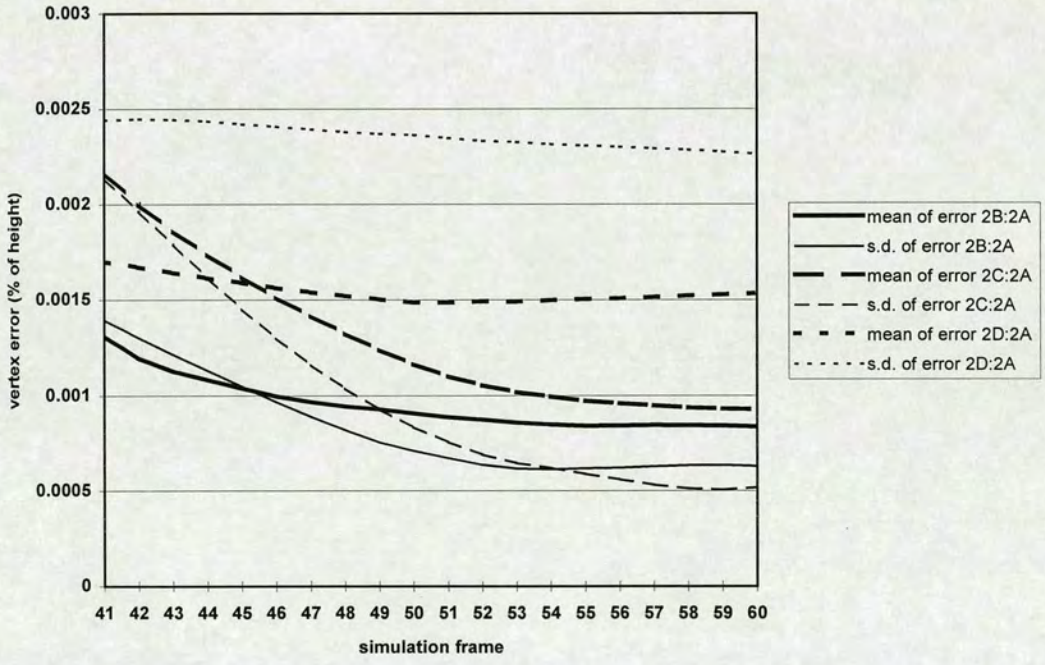



Figure 5.9(b): Accuracy comparison (scenes *2B*, *2C*, *2D* against scene *2A*)

## 5.10 Results

To give an indication of the visible differences between the results of the example simulations, Fig. 5.10(a) shows the final frames of the simulations for scenes *1A*, *1B*, *1C* and *1D*, while Fig. 5.10(b) shows the final frames of the simulations for scenes *2A*, *2B*, *2C* and *2D*.

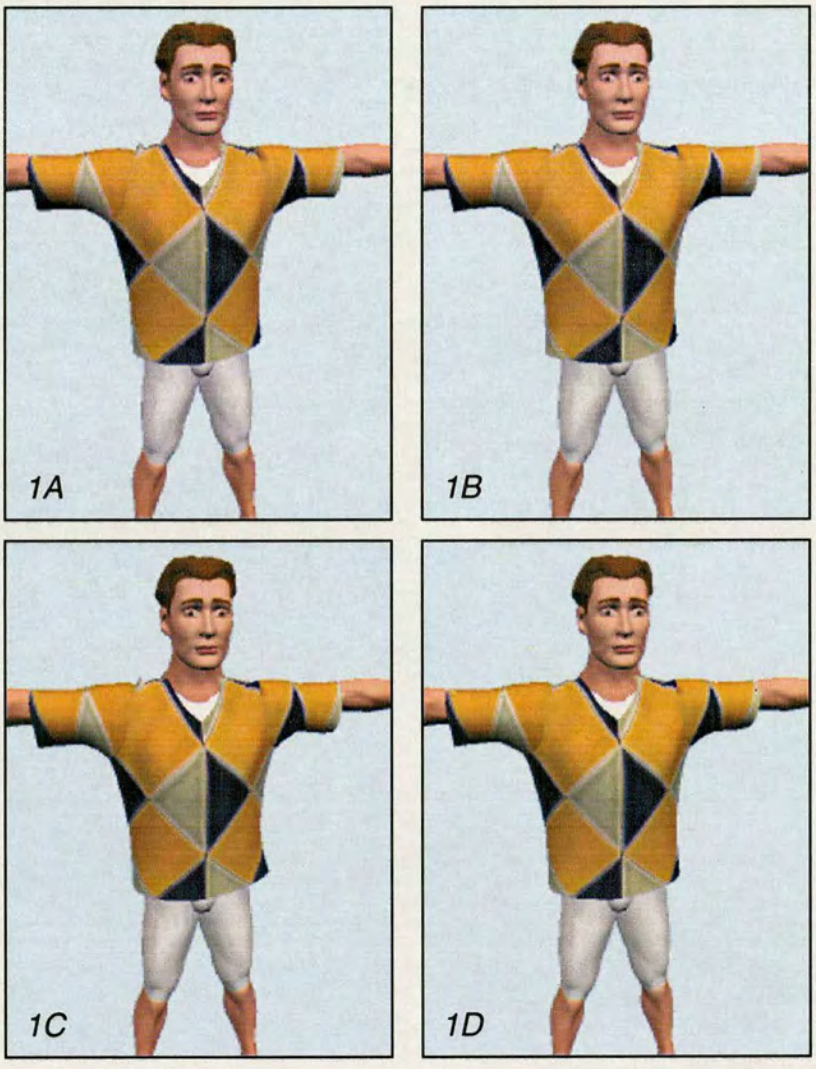



Figure 5.10(a): Final frames of simulations for scenes *1A*, *1B*, *1C* and *1D*


Figure 5.10(b): Final frames of simulations for scenes *2A*, *2B*, *2C* and *2D*

## 5.11 Conclusions

The timing data shown in Tables 5.2, 5.3 and 5.4 demonstrate that a valuable speed increase can be obtained by using progressive mesh garments in typical modelling scenarios. The progressive mesh method described here allows for flexibility in the degree of initial complexity and rate of progression used in any particular simulation. However, the accuracy analysis and the visual results provided for the example simulations show that even for extreme degrees and rates of progression, the error introduced into the final results can be quite negligible. Thus, the third point of the FIGMENT scheme provides an important contribution to the overall goal of reducing simulation times without compromising the fidelity of the final results.

## 5.12 Summary

This chapter has covered the third point of the FIGMENT scheme: the use of progressive meshes for garment modelling. After introducing the possibility of using reduced-complexity meshes for modelling, a number of methods for polygonal mesh decimation were briefly described and assessed for suitability. The previously developed technique of progressive meshes was argued to be highly appropriate for the present application, although requiring significant modification for that purpose. An algorithm for obtaining progressive mesh representations of high-complexity garment models was described, taking into account the special requirements and features of meshes which are to be used for dynamic physical-based simulation. The process of mesh reconstruction during simulation was also detailed, with special attention to the complexities introduced by the dynamic deformation of the meshes throughout. Finally, timing data, accuracy analysis and visual results from example simulations were given in order to illustrate the typical contribution of this point of the FIGMENT scheme towards the implementation of a mannequin service.

# Chapter 6

# The Hybrid Rendering Algorithm

## 6.1 Introduction

The preceding chapters have covered in detail the three points of the FIGMENT scheme devoted to optimizing the speed of modelling simulations: a simplified physical model, collision volume approximation and progressive meshes. Each of these points has been concerned with the geometry and dynamics of the meshes, without reference to the rendering of the garments and the surrounding modelling scene into a two-dimensional image to be displayed on the user's computer screen. The method of rendering the scene has been assumed to be perfect and independent of the other aspects of the modelling process.

The traditional algorithms for real-time (*i.e.* not ray-traced) rendering of polygon-based scenes generally fall into two categories: depth-sorting approaches and depth-buffering approaches. The latter is usually favoured for implementation in systems where memory constraints are not an issue. This approach, however, cannot provide acceptable results for clothing modelling applications unless stringent accuracy levels are placed on the depth-buffer itself (*i.e.* its precision) and, more seriously, on the collision handling algorithms implemented.

It would clearly be inappropriate for the FIGMENT scheme to abandon its efficient collision handling algorithms for the sake of visual consistency if an alternative solution can be found. This chapter therefore details a hybrid rendering algorithm which allows clothing items to be correctly rendered around a mannequin body even if there are minor interpenetrations present (either between cloth and body

or cloth and cloth). An additional advantage afforded by the algorithm is the ability to faithfully render multiple layers of clothing without implementing cloth-to-cloth collision handling.

The chapter first discusses the details of the basic OpenGL rendering algorithm, as used during the development of the FIGMENT scheme. Following this, the general principle of the hybrid rendering algorithm is presented, before detailing a particular implementation using the Open Inventor graphics library. The limitations of the algorithm are discussed and example results are shown which compare scenes modelled with and without the hybrid algorithm.

## 6.2 The OpenGL rendering algorithm

All computer graphics rendering algorithms must deal in some way with the problem of solid objects which partially or completely obscure other objects from the observer's point of view. Ray-tracing algorithms handle this issue directly; a light ray traced back from the viewpoint is reflected by the first object it meets in its path (although transparent and translucent materials complicate matters). Ray-tracing calculations are too time consuming for real-time rendering, however, and thus a simpler solution is required.

In the earlier days of computer graphics, when memory and processing power were limited, a depth-sorting algorithm could be used for modelled objects consisted solely of groups of polygonal faces. The entire set of polygons comprising the visible scene were sorted according to their distance (in some respect) from the viewpoint. The polygons were then rendered in order from the farthest to the nearest, the latter simply being superimposed over the former. Although efficient in terms of speed and memory requirements, the algorithm does not cope well with polygons that intersect, resulting in quite glaring inconsistencies in certain cases. Furthermore, as scenes increased in complexity, the time required to depth-sort the polygon set also increased until it reached unacceptable levels.

For these reasons, and since memory restrictions have become not nearly so pressing, the majority of real-time renderers use a depth-buffering algorithm.[1] The depth-buffer (or z-buffer) algorithm was originally developed by Catmull (1974) and is very simple to implement in either software or hardware. It requires that an additional memory buffer be allocated with entries corresponding to each pixel in the rendered image; hence for an 800 by 600 image, 480000 entries are needed. Before rendering an image, the depth-buffer is cleared by setting every entry to zero, which corresponds to the position on the *z*-axis of the back clipping plane (*i.e.* the farthest distance at which an object, or part of an object, will be rendered). The maximum value of each entry (which will depend upon the implementation) corresponds to the position on the *z*-axis of the front clipping plane (*i.e.* the nearest distance at which an object, or part of an object, will be rendered). During rendering, each polygon of the three-dimensional scene is scan-converted to project it onto the two-dimensional image; however, before writing each pixel to the image, the depth-buffer value corresponding to its position (as part of the polygon) in the *z*-axis is computed and compared to the current value in the depth-buffer for that pixel position. If that value is greater than the current value, *i.e.* the pixel is 'nearer' than that already present in the image, then the pixel is drawn in the image and the depth-buffer value is updated accordingly. Otherwise, the pixel is ignored, since it should not be visible in the final image. Thus, the set of polygons which comprise the scene may be rendered in any order whilst still ensuring that visible surfaces of the objects are as they should be: visible. In addition, intersecting polygons are also rendered appropriately.

The OpenGL graphics interface (Kempf and Frazier, 1997, Woo, Neider and Davis, 1997), developed by Silicon Graphics Inc., is one widely-used example of a depth-buffer implementation and provides the low-level rendering operations for the Open Inventor object-oriented 3D toolkit (the software implementation platform used during the development of the FIGMENT scheme). The OpenGL interface allows for either hardware or software implementations of the depth-buffer algorithm (depending on the hardware specification of the platform) and can be specified as

[1] A number use scan-line algorithms, however, as developed by Wylie *et al* (1967), Bouknight (1970) and Watkins (1970).

either 16-bit or 32-bit accuracy for each value in the buffer. However, as it stands, it provides less-than-satisfactory rendering results for an implementation of the FIGMENT scheme which includes only the three points detailed in previous chapters (simplified physical model, collision volume approximation and progressive meshes) as demonstrated by Fig. 6.1. Rather than seeking a solution which relies purely on adjusting the geometry of the scene, a modification of the OpenGL rendering algorithm which overcomes these rendering discrepancies, if possible, would provide a more efficient and elegant approach. For this reason, the FIGMENT hybrid rendering algorithm was developed and implemented using additional features of the OpenGL interface and by extending the Open Inventor library.

Figure 6.1: Example showing inadequacy of depth-buffered rendering

## 6.3 The FIGMENT hybrid rendering algorithm

The basis of the FIGMENT hybrid rendering algorithm was conceived by considering the specific characteristics of the scenes being modelled. Firstly, no part of the clothing should penetrate the mannequin body; therefore, no polygon of the cloth mesh should intersect with any polygon of the body, and thus depth-buffered rendering is not required for such intersecting polygons. Secondly, in the nature of the case, the clothing *surrounds* the body, which means that those polygons of cloth meshes which face away from the observer (with respect to their surface normals) will tend to be further away from the observer than the polygons of the body with which they come in to contact (and by which they should be obscured if in the same line of view). Conversely, those polygons of cloth meshes which face towards the observer tend to be nearer to the observer than the polygons of the body with which they come in to contact (and which should be obscured by the mesh polygons if in the same line of view).

These characteristics suggest the suitability a modified rendering algorithm based on a combination of both depth-buffering and depth-sorting techniques. When using a pure depth-buffering approach, the rendering discrepancies occur in those cloth mesh polygons at the 'front' of the scene (with respect to the viewpoint). A depth-sorting algorithm would theoretically ensure that these polygons were rendered correctly by obscuring those 'behind' them, *i.e.* those of the mannequin body. However, depth-sorting algorithms usually sort with respect to some approximation of the polygon's position, *e.g.* its centre-point, and in practice this means that certain orientations and relative sizes of polygons could cause the algorithm to fail, resulting in even worse rendering discrepancies. A better approach, therefore, is to make assumptions about the *relative* depth of the polygons (*i.e.* the order with respect to depth) by considering some other aspect of the polygons, *e.g.* the extent to which they face towards the observer. If a cloth mesh polygon faces towards rather than away from the observer (*i.e.* the angle between its surface normal vector and the direction of the viewpoint from the polygon is less than $90^{\circ}$) then it can be assumed to lie 'in front of' the body polygons with which it is in close proximity.

The basic FIGMENT rendering algorithm therefore proceeds by computing the extent to which each polygon in a cloth mesh faces the viewpoint and *forces* those facing the viewpoint to be rendered 'in front' of the body polygons. A measure of the extent to which a polygon faces the viewpoint is given by the vector dot product of its normal vector and the direction of its centre-point from the viewpoint:

$$k = \overline{n} \cdot \frac{(\overline{v} - \overline{c})}{|\overline{v} - \overline{c}|} \tag{6.1}$$

where $\overline{n}$ is the normal vector, and $\overline{v}$ and $\overline{c}$ are the position vectors of the viewpoint and centre-point of the polygon, respectively. Thus a constant $C$ may be specified, such that when $k \geq C$ the polygon will be rendered 'in front' of all previous polygons in the rendering pipeline. This constant corresponds to the value of $\cos\theta$, where $\theta$ is the angle between the polygon's normal vector and the direction of the viewpoint greater than which the polygon should be rendered normally, *i.e.* taking the depth-buffer into account, rather than being rendered 'in front'. If $C = 0$ then all polygons facing more towards than away from the viewpoint will be rendered 'in front'; in practice, a value slightly greater than zero produces more favourable results.

This basic approach solves the problem of cloth polygons which appear to penetrate the body of the mannequin. However, a further refinement to the algorithm is required. Cloth meshes which form folds which obscure (completely or partially) other parts of the mesh can be rendered incorrectly if those polygons obscured by the fold are rendered subsequent (in the rendering 'pipeline') to those comprising the fold. The polygons, though obscured by other mesh polygons, are technically facing the viewpoint and are therefore rendered 'in front' of all previously processed polygons. For this reason, the FIGMENT rendering algorithm requires that the polygons of a cloth mesh be depth-sorted before rendering the polygons, in order to guarantee that any polygons obscured by folds in the cloth are rendered prior to those comprising the folds.

The following pseudo-code summarizes the basic rendering algorithm for the polygons of a cloth mesh:

```
Polygon ordered_list[];
foreach ( polygon in cloth mesh ) {
   d = distance of polygon from viewpoint;
   addToOrderedList( polygon, d, ordered_list );
}
foreach ( polygon in ordered_list ) {
   k = extent to which polygon faces viewpoint;
   if ( k ≥ C )
      setDepthBufferCondition( ignore );
   else
      setDepthBufferCondition( normal );
   render( polygon );
}
```

It should be observed, of course, that for the hybrid rendering algorithm to provide the desired results, the cloth mesh polygons should enter the rendering 'pipeline' *subsequently* to the polygons of the mannequin body. Furthermore, any other components of the modelling scene, *e.g.* furniture or scenery, should be submitted to the rendering pipeline subsequently to the polygons of the cloth mesh in order to avoid being incorrectly obscured by those polygons. These considerations will almost certainly affect the structure of the scene graph if an objected-oriented graphics library, *e.g.* Open Inventor, is used.

The following section details the implementation of the FIGMENT rendering algorithm by extending the Open Inventor library. It also describes the implementation of several additional measures needed to avoid certain other problems introduced by the use of the basic hybrid algorithm.

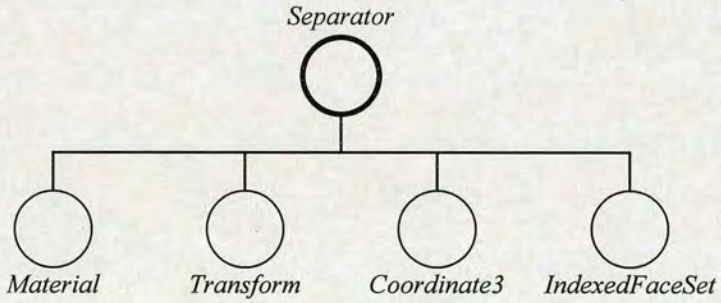



Figure 6.2: Typical Open Inventor scene graph

## 6.4 Extensions to the Open Inventor library

Open Inventor (Wernecke, 1994a) is an object-oriented 3D toolkit which provides a library of high-level data structures and functions for creating, manipulating and rendering three-dimensional scenes. The actual rendering of the scenes is performed using the OpenGL graphics library. Modelling scenes exist as hierarchical scenes graphs comprised of 'nodes' which generally either represent geometrical objects in the scene or specify information which modifies the attributes, geometry and behaviour of nodes which follow it within that scene graph. A simple Open Inventor scene graph is shown in Fig. 6.2. It consists of a *Separator* node which acts as a 'parent' to group together 'child' nodes within the scene; a *Material* node which specifies the material attributes (*e.g.* diffuse colour) to use when rendering geometrical objects; a *Transform* node which specifies a geometrical transformation to be applied to objects; a *Coordinate3* node which specifies a set of three-dimensional vertex coordinates; and an *IndexedFaceSet* node which specifies a set of polygons comprising an object which should be drawn during the rendering process. The Open Inventor library also provides a number of 'actions' which can be applied to a scene graph—the most significant being the 'rendering action'—which take as an input the 'root' node of the graph and process the nodes in a top-down left-to-right order. The children of a grouping node are processed before any sibling nodes to the right of it. Thus, the *Material* and *Transform* nodes modify the appearance and geometry of the *IndexedFaceSet*, and the *Coordinate3* node specifies the basic vertex positions for the polygons of the *IndexedFaceSet*. The *Separator* node acts to group these four nodes together and also to shield any successive sibling nodes (and their children) from the effects of its child nodes.

An *IndexedFaceSet* node is therefore used to represent a polygonal object in the scene graph and would normally be used for both the body parts of the mannequin and the cloth meshes of the garments in a modelling scene. As a node, it possesses a number of 'fields' which specify its actual characteristics. Of most importance is the *coordIndex* field, which consists of a list of vertex indices, corresponding to the vertices of a preceding *Coordinate3* node, which defines the individual polygons of

the object. The value -1 is used as a delimiter. The following node specification (using the Open Inventor scene graph file format) defines the geometry of a pyramid object:[2]

```
# Coordinate3 node defines five vertices
Coordinate3 {
   point [ 1 0 1, 1 0 -1, -1 0 -1, -1 0 1, 0 1 0 ]
}
# IndexedFaceSet node defines four triangular
# polygons and one square polygon
IndexedFaceSet {
   coordIndex [
      0, 1, 4, -1,
      1, 2, 4, -1,
      2, 3, 4, -1,
      3, 0, 4, -1,
      3, 2, 1, 0, -1
   ]
}
```

By convention, the 'outer' face of a polygon is defined as that observed by an *anticlockwise* ordering of vertices, regardless of whether one or both sides of a polygon are to be visible. This ordering of vertices also determines the direction of the normal vector for a planar polygon.

The actual implementation of the *IndexedFaceSet* node is relatively simple. When an *IndexedFaceSet* node is encountered during the scene graph traversal of a rendering action, a set of polygons is compiled from the fields of the node in combination with the current 'traversal state'. The traversal state is the set of properties (*e.g.* material attributes, cumulative geometrical transformation, vertex coordinate set, *etc.*) which have been specified or modified from their default values by the preceding nodes in the scene graph. Once a list of polygons with the correct material attributes and vertex positions has been compiled, each polygon is submitted to the OpenGL rendering pipeline, in the order specified in the *coordIndex* field of the node. In order to implement the FIGMENT hybrid rendering algorithm, however, an alternative to the basic *IndexedFaceSet* node is required. Fortunately, the Open Inventor toolkit allows for (and actively encourages) the development of customized nodes for inclusion within scene graphs (Wernecke, 1994b).

[2] In the Open Inventor file format, comments are preceded by a '#' character and are ignored during parsing.

The *ClothingLayer* node

The *ClothingLayer* node, an extension to the Open Inventor node library described in this section, provides an alternative to the *IndexedFaceSet* node which uses the hybrid rendering algorithm described in the previous section. Possessing the same fields as the *IndexedFaceSet* node, with each field having the same significance, it is intended as a direct replacement for the latter node when using garment models in a FIGMENT-based clothing simulation. However, the *ClothingLayer* node has an additional field, *criticalCosineAngle*, corresponding to the constant $C$ defined previously and used to specify which polygons in the cloth mesh are taken to be 'in front' of the mannequin body.

Thus, all *IndexedFaceSet* nodes within a clothing item model should be replaced with appropriate *ClothingLayer* nodes. It must also be ensured that, in the overall scene graph, these nodes will be traversed *after* those which represent the mannequin body but *before* those which represent other objects in the scene (if present) for the reasons previously explained.

Sample implementation code for the *ClothingLayer* node is given in Appendix C.

Figure 6.3: Example of rendering discrepancies due to poor ordering of *ClothingLayer* nodes

The *DepthSortedGroup* node

Using the *ClothingLayer* node alone may still result in highly unsatisfactory results for multiple-mesh clothing items. For example, Fig. 6.3 shows a mannequin clothed with a shirt comprised of six individual cloth meshes which are seamed together; the traversal order of the six *ClothingLayer* nodes in the scene graph has resulted in polygons of one mesh incorrectly obscuring those of another. In order to render the scene correctly, it should be estimated which meshes are further away from the viewpoint, *i.e.* potentially behind other meshes, and should therefore be rendered *before* those meshes. For this reason, another customized node may be implemented to take care of the multiple-mesh problem.

The *Group* node, included within the Open Inventor node library, provides the basic functionality for grouping together nodes within a scene graph. An instance of the node may possess any number of children (or none at all) which are traversed in indexical order, *i.e.* from left-to-right. (The *Group* node differs from the *Separator* node in that it doesn't act to shield any successive sibling nodes (and their children) from the effects of its child nodes.)

The *DepthSortedGroup* node is derived from the *Group* node with the substantial difference that its child nodes are traversed in order according to their depth in the scene (*i.e.* distance from the viewpoint) rather than simply in indexical order. The depth-sorting is done with respect to the centre-point of the bounding box of each child (sub-tree). Computing the bounding box of each child can be time consuming and need not necessarily be performed prior to every rendering action; if the viewpoint is fixed and the clothing meshes do not change with respect to their depth order during the simulation, then the computation only needs to be done once. For this reason, the computation of the centre-points is not done automatically prior to each rendering pass but by calling a method, *recomputeCentres*, implemented for the *DepthSortedGroup* object (*i.e.* the representation of the node in the object-oriented implementation language). This method must be called again if the view of the modelling scene changes substantially during the simulation.

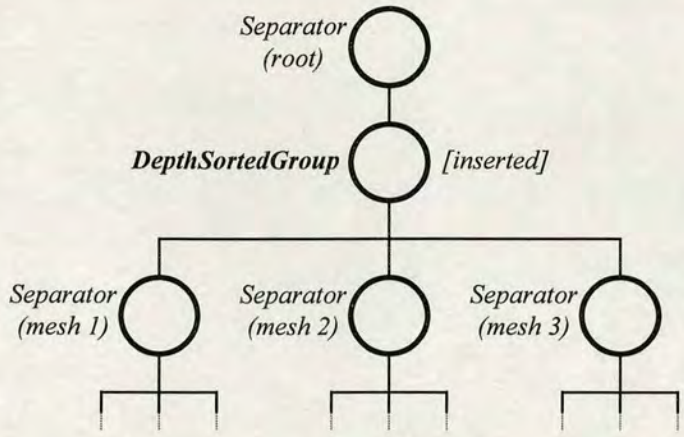



Figure 6.4: Insertion position for *DepthSortedGroup* node

The *DepthSortedGroup* node is utilised by inserting it into the scene graph between the parent node of the sub-graphs which comprise the individual cloth meshes and that node's immediate children (Fig. 6.4).

Sample implementation code for the *DepthSortedGroup* node is given in Appendix C.

The *StencilSeparator* and *StencilClear* nodes

The implementation of the *ClothingLayer* and *DepthSortedGroup* nodes detailed above allow multiple-mesh clothing items to be satisfactorily rendered such that *(a)* the meshes do not appear to penetrate the mannequin body, *(b)* the 'folds' of individual meshes are not obscured by polygons directly behind them, and *(c)* meshes are not obscured by those directly behind them. Another rendering discrepancy can occur, however, for certain poses of the mannequin. For example, a pose which involves an exposed limb of the mannequin appearing in front of a clothing item will be incorrectly obscured by the polygons of that item (Fig. 6.5).

In order to avoid this problem, while still retaining the advantages of the *ClothingLayer* and *DepthSortedGroup* nodes, further extensions may be made to the Open Inventor node library which take advantage of the OpenGL 'stencil' buffer facility. The stencil buffer is a means by which certain areas of the image being rendered may be 'masked' whilst drawing the rasterized polygons. It consists of an

array of (usually binary) values with the same dimensions as the image; during a rendering pass, the value of an entry in the buffer can be tested to determine whether the corresponding pixel in the image may be replaced. Usually, the OpenGL stencil buffer is ignored during the rendering of an Open Inventor scene.

Figure 6.5: Example of clothing incorrectly obscuring mannequin

Figure 6.6(a): Object parented by *StencilSeparator* node

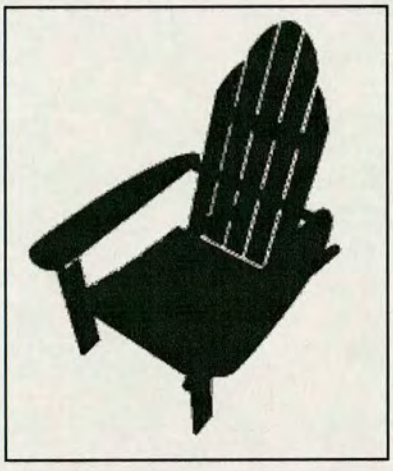


Figure 6.6(b): Resultant mask in stencil buffer

What is required, then, is a method of masking the part of the mannequin body which is being incorrectly obscured. This can be effectively achieved by the use of two additional nodes: *StencilClear* and *StencilSeparator*.

The *StencilClear* node has no effect within a scene graph except for when it is encountered during the traversal of a rendering action, in which case it simply acts to clear the stencil buffer. It is intended to appear early on in the traversal order of a scene graph, prior to the traversal of any *StencilSeparator* nodes.

The *StencilSeparator* node is derived from, and acts as a direct replacement for, the basic *Separator* node of the Open Inventor library. In addition to acting as a grouping parent for sub-graphs and isolating the rest of the scene graph from any changes in the traversal state effected by its children, it also results in a mask corresponding to the shape of its sub-graphs (as rendered two-dimensionally) being written to the stencil buffer during a rendering action. For example, the sub-graph which represents the object in Fig. 6.6(a) if parented by a *StencilSeparator* node would result in the stencil buffer shown in Fig. 6.6(b).

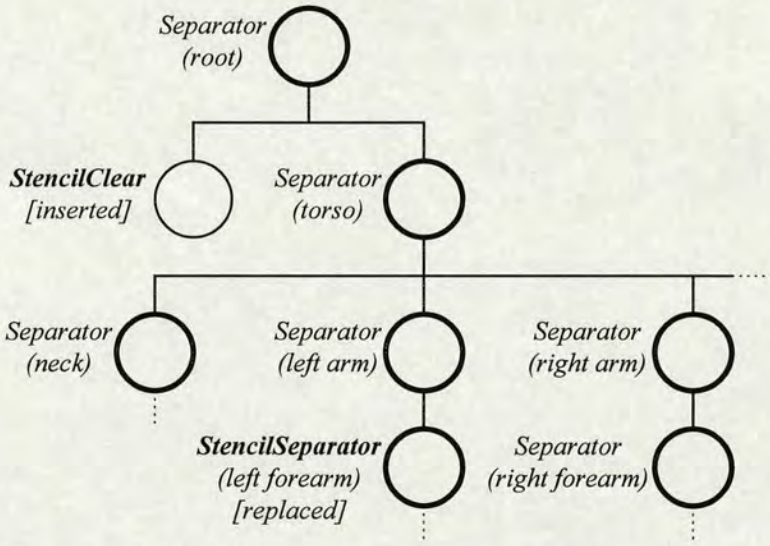



Figure 6.7: Position of *StencilClear* and *StencilSeparator* nodes within scene graph

The resultant mask in the stencil buffer may thus be used in the rendering of cloth meshes to avoid mannequin body parts being incorrectly obscured. The rendering method of the *ClothingLayer* must be amended such that the stencil buffer is no longer ignored in the rendering of the polygons but is used to prevent those pixels in the image which correspond to non-zero values in the stencil buffer from being overwritten. In addition, a *StencilClear* node must be inserted into the scene graph at an appropriate place, and *StencilSeparator* nodes must be inserted (or must replace

existing *Separator* nodes) to enclose the sub-graphs which represent the relevant body parts (Fig. 6.7). The modelling scene of Fig. 6.5 rendered with the appropriate *StencilClear* and *StencilSeparator* nodes is shown in Fig. 6.8.

Sample implementation code for the *StencilClear* and *StencilSeparator* nodes is given in Appendix C.

Figure 6.8: Example of using *StencilClear* and *StencilSeparator* nodes

## 6.5 Advantages and limitations

An effective implementation of the FIGMENT hybrid rendering algorithm should include the additional cloth mesh sorting and stencil buffering techniques described above in the Open Inventor *ClothingLayer*, *DepthSortedGroup*, *StencilClear* and *StencilSeparator* nodes. The primary advantage of the hybrid rendering algorithm is clearly that for which it was developed, *i.e.* to avoid undesirable rendering discrepancies without resorting to complex and time-consuming geometry manipulating methods. An additional important advantage of the algorithm, however, is that it also allows multiple layers of clothing to be adequately rendered without the need for cloth self-collision detection (which would also require complex and time-consuming algorithms to implement). For example, a shirt which hangs down lower than the waistline of a pair of trousers correctly appears to lie fully inside of those trousers, while a jacket worn over the shirt correctly appears to enclose both items (Fig. 6.9). For more complex items, features such as pockets and lapels are handled appropriately (Fig. 6.10). In the cases of multiple clothing items, the scene graph must simply be arranged such that outside garments occur later in the traversal order of the graph than inner garments (Fig. 6.11).

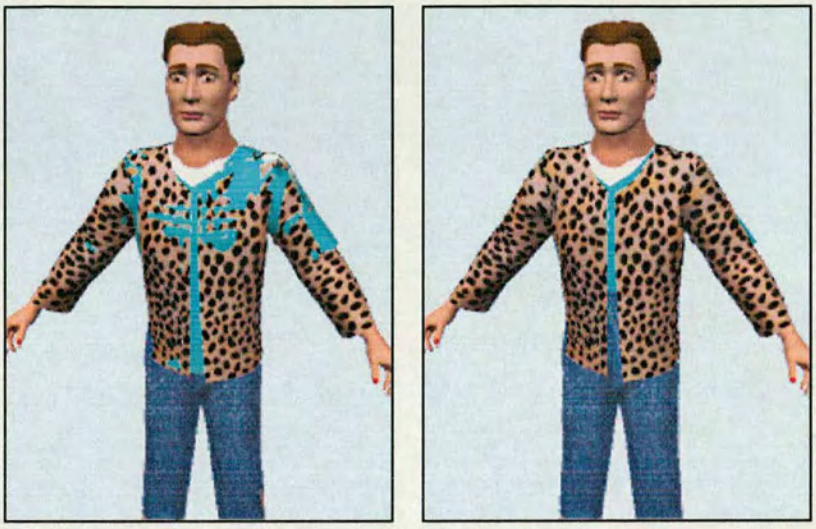


Figure 6.9: Example with multiple layers of clothing
(without and with hybrid rendering algorithm)

Figure 6.10: Example of shirt with lapels
(without and with hybrid rendering algorithm)

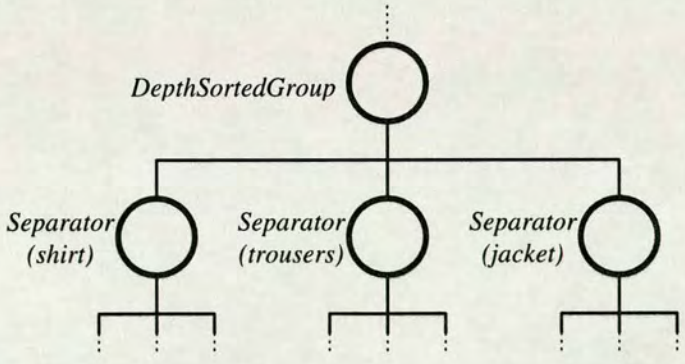



Figure 6.11: Order of multiple overlapping garments within scene graph

There are a number of limitations in the use of the hybrid rendering algorithm, however. Firstly, the depth sorting algorithms will inevitably increase the computation time when rendering a scene. The following section details the extent of this in a typical implementation.

Secondly, there are certain poses and views of a mannequin which cannot be rendered satisfactorily using any combination of the Open Inventor nodes detailed above. These generally involve parts of the mannequin which are enclosed by one cloth mesh, *e.g.* a forearm, which should also obscure other cloth meshes which occur in the same line of view. Fig. 6.12 illustrates one example of this. One possible solution will be offered here, although implementation details must be left as an exercise for the reader. The most straightforward answer is to allow selectivity of which cloth meshes should use the stencil buffer or not. In Fig. 6.12, for example, the main body of the shirt *should* use the stencil mask created from the mannequin forearm, whereas the sleeve of the shirt which surrounds the forearm *should not*. In this instance, only a binary selectivity is required; in more complex scenes however, multiple *layers* of stencils may be required depending on the pose of the mannequin. The OpenGL interface allows the stencil buffer to hold multiple-bit values and arithmetical comparison operations may be used to determine the masking function during rendering, so the implementation of a multiple layer stencil ought not to be too difficult in this respect. The complexity will result from developing an algorithm to depth-sort the various elements in the modelling scene, and to determine *(a)* which body parts will contribute to each stencil layer and *(b)* which layer is appropriate for the rendering of each cloth mesh. If the pose of the mannequin or the scene viewpoint change substantially then the algorithm must be reapplied. The advantage of this suggested solution is that the nodes comprising the mannequin body and the nodes comprising the cloth meshes may still be kept separate within the scene graph; in other words, the mannequin body may still fully precede the clothing items in the rendering pipeline.

Figure 6.12: Stencilled body part incorrectly obscures other meshes

Figure 6.13: Inner layer appears to protrude from outer layer

A third limitation of the rendering algorithm can be seen in certain modelling scenes in which clothing items which should be physically constrained in some places by other clothing items, although correctly rendered 'inside' those items, still appear to protrude from behind them due to the absence of cloth-to-cloth collision detection. An example of this phenomenon is shown in Fig. 6.13. A simple solution to this problem might involve simulating forces of attraction between points on the mannequin and points on the inner cloth meshes; for example, in Fig. 6.13, forces which pull the bottom of the shirt in towards the waistline would be appropriate.

## 6.6 Results

The rendered results of two typical modelling scenarios using the FIGMENT hybrid rendering algorithm are shown in Fig. 6.14 and Fig. 6.15. Table 6.1 indicates the relative increase in rendering time and the relative increase in overall simulation time in each case. The simulations were run on a 200MHz Pentium PC platform.

Figure 6.14: Example scene *A*

Figure 6.15: Example scene *B*

| Scene | Number of simulation iterations | Number of iterations between rendered frames | Percentage increase in rendering time using hybrid algorithm | Percentage increase in simulation time using hybrid algorithm |
|---|---|---|---|---|
| *A* | 6000 | 100 | 36 % | 5.5 % |
| *B* | 6000 | 100 | 62 % | 4.7 % |

Table 6.1: Computational cost of hybrid rendering algorithm

## 6.7 Conclusions

The results above, and the other rendered images featured in this chapter, demonstrate that the hybrid rendering algorithm provides an effective solution to the problem of rendering discrepancies due to the inadequacy of the standard depth buffer approach. Not only are these discrepancies avoided, but minor inaccuracies in the physical modelling process are also compensated for. Furthermore, multiple layers of clothing may also be effectively modelled without resorting to the implementation of computationally costly cloth-to-cloth collision handling algorithms. Although a minor computational cost is invoked by the sorting stages of the hybrid algorithm, in a typical implementation this increase is marginal in terms of the overall simulation time.

Although there are a number of limitations involved in the use of the hybrid algorithm as described here, problem cases may be avoided either by the careful choice of modelling poses and viewpoints or by the further development of the algorithm and the accompanying software in the ways briefly suggested above.

## 6.8 Summary

The fourth and final point of the FIGMENT scheme—a hybrid rendering algorithm—has been shown in this chapter to be an invaluable component in the formulation of an approach to interactive garment modelling. The standard rendering approaches to the 'hidden surface problem', although efficient and suitable for general purposes, are found to be inadequate for the present application. For this reason, a rendering algorithm which combines the advantages of two standard algorithms has been shown to avoid the rendering discrepancies present in results rendered by the standard algorithms. The chapter has discussed the implementation of the algorithm using the OpenGL graphics interface and in the form of node extensions to the Open Inventor graphics library, with accompanying node extensions to overcome the major problems introduced by the use of the hybrid algorithm. The remaining limitations of this approach to rendering (in terms of the

proposed implementation) have been mentioned, with brief suggestions for further development of this implementation of the algorithm in order to remove these minor limitations. Finally, examples of typical modelling scenes rendered using the hybrid algorithm were provided, along with corresponding timing data to demonstrate the relatively minor computation cost introduced by applying this point of the FIGMENT scheme—a cost which is more than compensated for by the advantages of using the algorithm.

# Chapter 7

# The FIGMENT Scheme in Practice

## 7.1 Introduction

In Chapters 2 through 6 of this thesis, the technical details of the FIGMENT scheme have been described, along with empirical timing and accuracy data obtained from typical garment modelling simulations which demonstrate the superiority of the FIGMENT modelling and rendering algorithms. In this chapter, the results of a full implementation of the FIGMENT scheme are discussed; in particular, the implementation in practice of a FIGMENT-based mannequin service is considered.

The first section of this chapter provides timing data to illustrate the speed gains possible for typical clothing simulations when using all four points of the scheme together and under various configurations. Following this, the majority of the chapter is concerned with discussion of a user experiment, the aim of which was to assess whether a fully FIGMENT-based implementation of the service was judged to be preferred by users on the basis of its speed and without any perceived loss of quality or accuracy. After detailing the methodology, procedure, and implementation details of the experiment, the results of the experiment are presented and analysed.

A brief discussion of the possibility of a FIGMENT-based mannequin service which uses the *Virtual Reality Modelling Language* (VRML) is given before concluding the chapter.

## 7.2 Combined results

In this section, timings results are provided for two typical modelling simulations to illustrate the speed gains possible when using a full implementation of the FIGMENT scheme under various configurations.

The two modelling scenes were simulated on a Pentium II 233MHz PC using six different combinations of FIGMENT features. Scene *A* consisted of a male mannequin (21,422 polygons) clothed with a T-shirt (1,723 polygons) and a pair of trousers (1,330 polygons); scene *B* consisted of a female mannequin (22,190 polygons) clothed with a dress (1,103 polygons). The six combinations are detailed in Table 7.1. As described in Chapters 3 and 4, the 'octree' collision method refers to a polygon-based hierarchically-optimised detection algorithm, the details of which are provided in Appendix A.

| Simulation | Time-step | Instability-countering measures | Collision handling method | Progressive meshes | Physical model |
|---|---|---|---|---|---|
| 1 | 0.1 ms | no | octree | no | standard |
| 2 | 1.0 ms | yes | radial depth | no | standard |
| 3 | 1.0 ms | yes | capsule | no | standard |
| 4 | 1.0 ms | yes | radial depth | yes | standard |
| 5 | 1.0 ms | yes | capsule | yes | standard |
| 6 | 1.0 ms | yes | capsule | yes | fast |

Table 7.1: Simulation specifications

Table 7.2 details the timing results obtained by running the six simulations for each scene. The final frame of each simulation of scene *A* is shown in Fig. 7.1; the final frame of each simulation of scene *B* is shown in Fig. 7.2. These images illustrate that the visible distortions resulting from the use of the FIGMENT algorithms are minimal and would in no way hinder the usability of a FIGMENT-based virtual mannequin service—a service which would be impossible without such effective speed-enhancing techniques.

| Simulation | Duration (secs) | Average % of total computation devoted to internal forces | Average % of total computation devoted to collision handling | Simulation speed gain with respect to simulation 1 |
|---|---|---|---|---|
| *A1* | 10795 | 17.8 | 67.7 | — |
| *A2* | 345 | 54.9 | 33.4 | 28.3 |
| *A3* | 316 | 59.4 | 28.1 | 34.2 |
| *A4* | 204 | 50.2 | 34.6 | 52.9 |
| *A5* | 182 | 54.3 | 29.1 | 59.3 |
| *A6* | 137 | 38.6 | 39.4 | 78.8 |
| *B1* | 4362 | 15.8 | 68.8 | — |
| *B2* | 116 | 59.5 | 30.3 | 37.6 |
| *B3* | 117 | 56.9 | 33.3 | 37.3 |
| *B4* | 73 | 53.4 | 32.9 | 59.8 |
| *B5* | 72 | 52.0 | 34.6 | 60.6 |
| *B6* | 55 | 36.2 | 46.2 | 79.3 |

Table 7.2: Timing results for example simulations

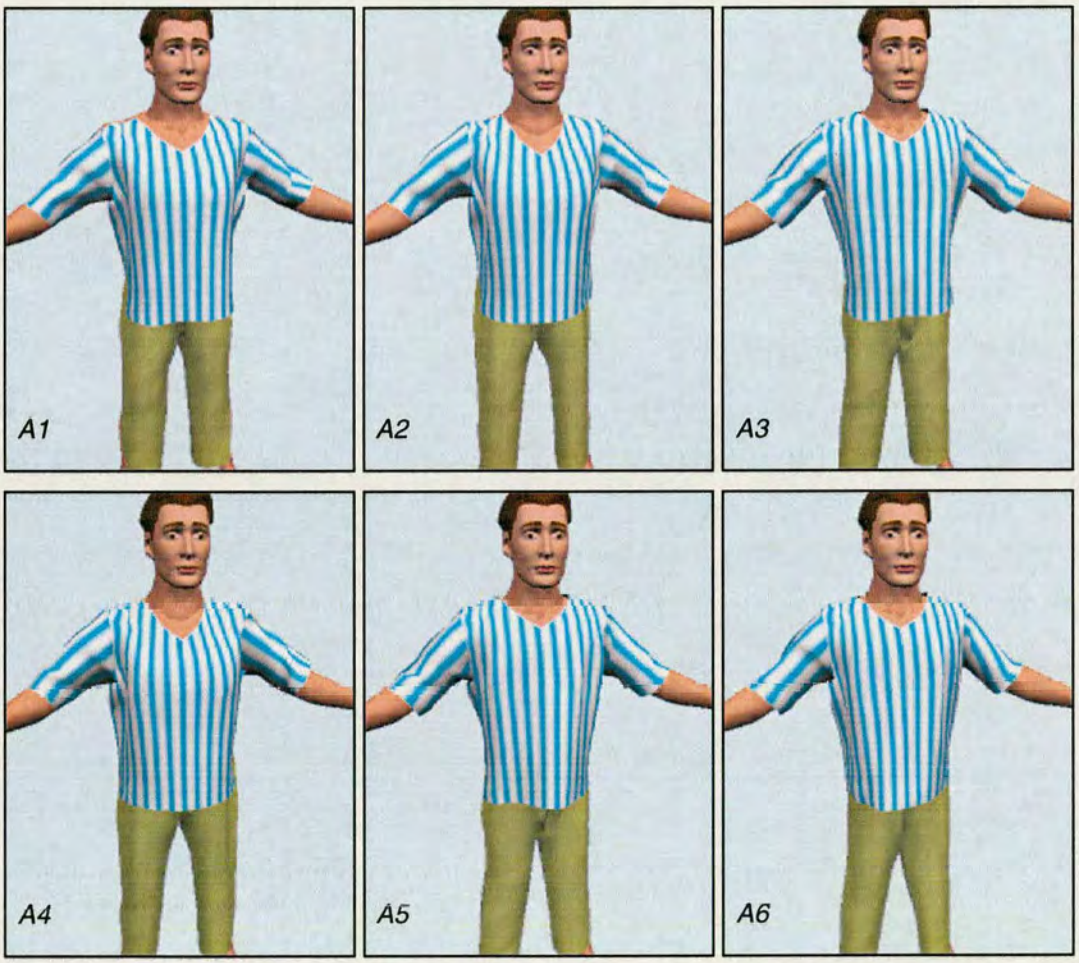



Figure 7.1: Final frames output from example modelling simulations (scene *A*)

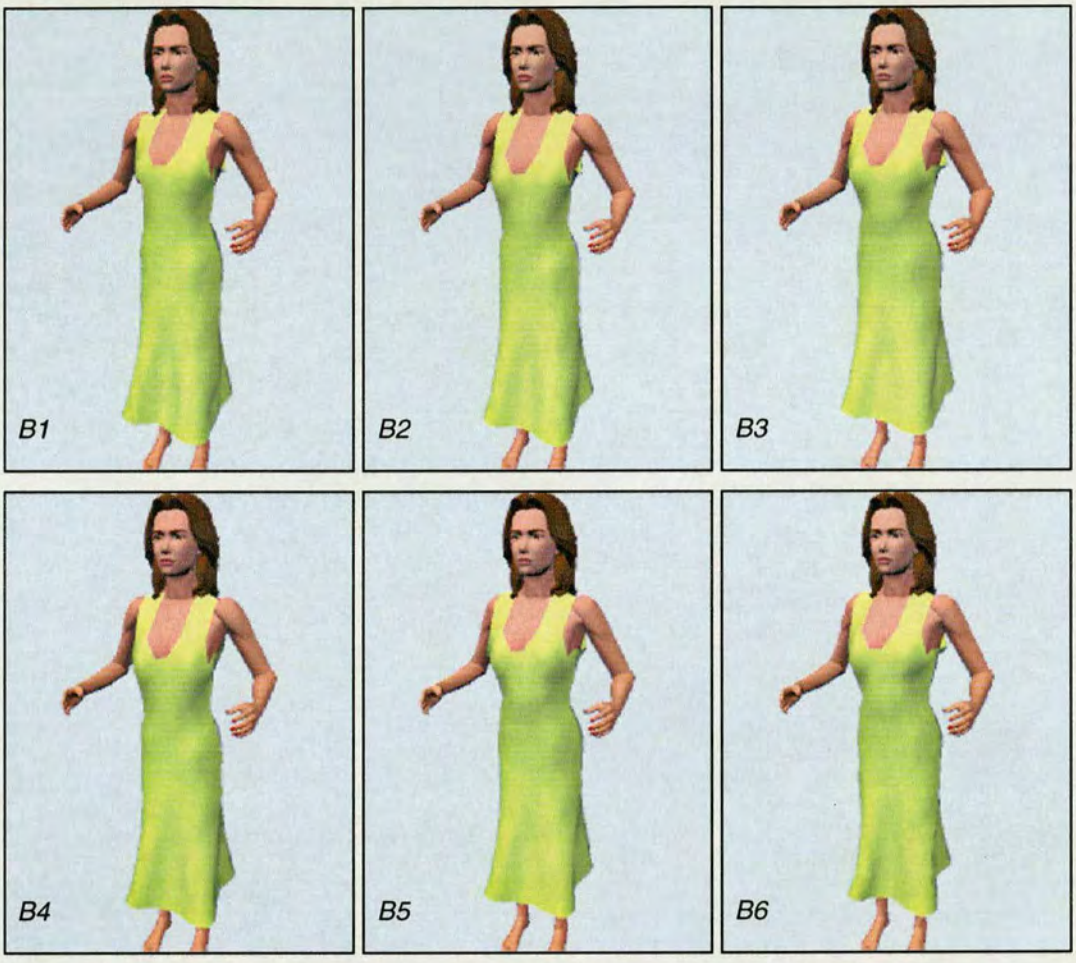



Figure 7.2: Final frames output from example modelling simulations (scene *B*)

## 7.3 Experiment methodology

This section begins the discussion of the user experiment designed to assess user attitudes to a FIGMENT-based mannequin service. The hypothesis for the experiment described here may be stated thus:

> *Users of a virtual fitting room will prefer to use a fully FIGMENT-based service rather than a representative non-FIGMENT-based service on account of the increased speed of the former, while not perceiving any loss of fidelity or quality in the former.*

In order to establish the hypothesis, the following methodology for the experiment was formulated. A representative sample of the public would be asked to try both a fully FIGMENT-based service and a non-FIGMENT-based service[1] in a simulated virtual shopping environment. The preference of each user would be assessed by asking them to make a free choice between the two services after having experienced both. In order to assess the basis for any predominant preference, two subjective measures would be obtained: firstly, a user attitude questionnaire would be completed by each participant after experiencing each service (but before expressing a preference); secondly, a final questionnaire would be completed to elicit the basis for the expressed preference.

The user attitude questionnaires would consist of statements (equally distributed for positive and negative wording) with which the users would indicate their agreement on a 7-point Likert-style scale (refer to Appendix D for an account of the Likert questionnaire format). The final questionnaire would list of a number of potential reasons for the users' preferences and ask them to indicate which applied in each individual case. The questionnaire would also allow an open response for each user to specify additional reasons, if desired, to those specifically listed.

## 7.4 Experiment procedure

The user trial was performed with 50 participants of distributed gender and age. The experiment methodology described above was embodied in the following procedure for each participant.

[1] As indicated in the following section, the non-FIGMENT-based service would, in fact, be a FIGMENT-based service with the main speed-enhancing algorithms—collision approximation and mesh progression—disabled. The reason for this decision is that these two points of the FIGMENT scheme are of most interest in assessing the relative usability of the two services. Implementing a more sophisticated physical model for the non-FIGMENT-based service and using a shorter time division for iteration would slow the service down to a literally unusable point, making any comparison between the services trivial. Similarly, disabling the hybrid rendering algorithm in that same service would render any comparison between the fidelity of the services quite meaningless since rendering discrepancies would inevitably appear in practice.

The participant was primed by reading the following description of the experiment:

> Your task in this experiment is to use a virtual clothing shop to buy an item of clothing for yourself.
>
> To help you know whether the clothing will fit you and suit you, you will use a virtual mannequin to try on the clothing. A virtual mannequin is a computer-generated human model customised by you to match your physical dimensions and appearance. The mannequin will then be able to try on clothing on your behalf.
>
> Once you have customised your own personal mannequin, you will take clothes into the fitting rooms provided in the shop. There are two fitting rooms – RED and BLUE – and you will be asked to use both of them.
>
> In the fitting rooms, the mannequin will put on the clothing you chose in the shop. In each fitting room, the clothing will be placed over the mannequin and, as you watch, it will slowly hang down and settle over the mannequin's body. As the clothing settles, you will be able to see how well the clothing fits you and also how it looks on you and whether it suits you.
>
> You can spend as much time as you like in each fitting room, watching the mannequin being dressed with the clothing. When you have seen enough you can leave the fitting room and continue shopping.
>
> The experiment supervisor will guide you through the experiment and will tell you exactly what to do at each stage.

The participant began by navigating a simple virtual clothes shop containing nine items of clothing (three styles in three sizes) hanging from rails (Fig. 7.3). The participant was instructed to select one item to 'try on' by clicking on it with the mouse pointer, and then to enter the male or female (virtual) fitting rooms by clicking on the appropriate doors (Fig. 7.4).

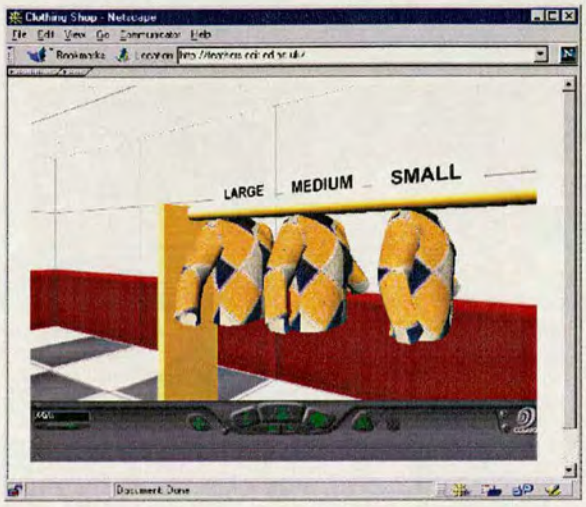



Figure 7.3: The virtual clothes shop

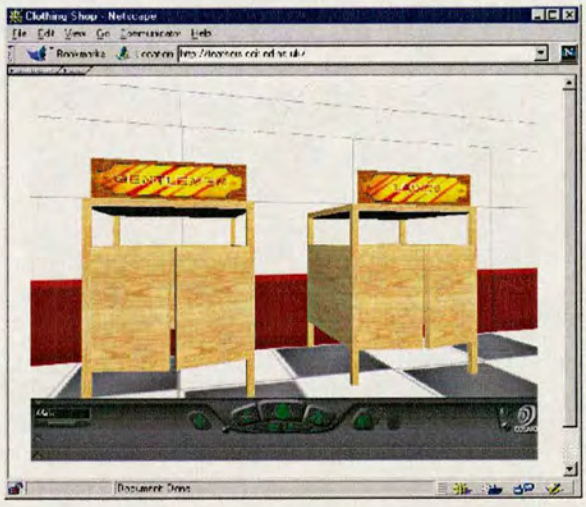


Figure 7.4: Fitting rooms within the virtual clothes shop

On entering the virtual fitting rooms, the participant was presented with an interface to allow him/her to customise a 'virtual mannequin' to match his/her physical dimensions and skin colour (Fig. 7.5). The interface allowed the adjustment of the mannequin dimensions both by specifying standard clothing sizes[2] or by adjusting individual body parts for a more precise specification. On exiting the interface, participants were required to confirm that the clothing sizes of the mannequin were correct before continuing.

[2] Male participants specified height, chest, waist, inside leg and coat length measurements. Female participants specified height, bra size, waist, hips and inside leg measurements.


Figure 7.5: The mannequin customisation interface


Figure 7.6: The 'red' and 'blue' fitting rooms


Figure 7.7: Inside the fitting room

Having customised his/her virtual mannequin, the participant was then presented with the two fitting rooms (Fig. 7.6). At this point, half of the participants were instructed to enter the 'red' room and the other half instructed to enter the 'blue' room, by clicking on the doors of that fitting room.

On entering the prescribed virtual fitting room, the participant was presented with a series of numbered images showing the virtual mannequin being dressed with the previously selected clothing item (Fig. 7.7). These images were simulation frames rendered by one of two specifications of garment modelling algorithms:

Specification A

| | |
|---|---|
| *time division for one iteration:* | 0.001s (virtual time) |
| *iterations between frames:* | 100 |
| *physical model:* | FIGMENT-based |
| *collision detection/response:* | octree-based polygon-to-polygon (see Appendix A) |
| *clothing meshes:* | non-progressive (2103 polygons in total) |
| *rendering algorithm:* | FIGMENT-based |

Specification B

| | |
|---|---|
| *time division for one iteration:* | 0.001s (virtual time) |
| *iterations between frames:* | 100 |
| *physical model:* | FIGMENT-based |
| *collision detection/response:* | FIGMENT-based (capsule approximation) |
| *clothing meshes:* | progressive (from 549 to 2103 polygons in total) |
| *rendering algorithm:* | FIGMENT-based |

For approximately half of the participants (randomly distributed), the 'red' fitting room used simulation specification A and the 'blue' fitting room used simulation specification B; for the remainder of the participants, the simulations were reversed such that the 'red' fitting room used simulation specification B and the 'blue' fitting room used simulation specification A. This was done to eliminate any contamination

of the results due to the labelling, appearance, or on-screen position of the two fitting rooms.

The participant was guided by on-screen instructions to watch the 'dressing' of the mannequin as the clothing item slowly settled over its body. Once the participant had judged to have seen enough (*i.e.* to know whether the garment would fit and how it would look) he/she clicked a button to exit the fitting room. At this point, the participant was asked to complete a short questionnaire (reproduced in Appendix E) regarding their experience of the fitting room.

After exiting the fitting room, the participant was presented with the two fitting rooms again (Fig. 7.6) and instructed by the supervisor to enter the *other* room, again by clicking on the doors of that fitting room. In the same way as previously, the participant watched the mannequin being 'dressed' (using the alternative simulation specification this time) until he/she decided that he/she had seen enough and exited the fitting room. The participant was then asked to complete a second questionnaire (identical to the first).

At this point in the experiment, the participant was instructed to return to the virtual clothes shop (by clicking the button provided on-screen) and select a *different* clothing item to 'try on'. After doing so, the participant was asked to re-enter the fitting rooms.

After clicking on the male or female fitting room doors, the participant was immediately presented for the third time with the 'red' and 'blue' fitting rooms. This time, the participant was asked to freely choose which fitting room to enter in order to see the clothing item modelled by the mannequin.

After the participant had finished watching the mannequin being 'dressed' (using the same simulation as before for that room) and clicked the button to exit the fitting room, the experiment was completed by asking the participant his/her reasons for the choice which was made. This was done by interviewing the participant and completing a corresponding questionnaire (reproduced in Appendix E).

## 7.5 Implementation details

The experiment described above was implemented in the form of an Internet-based service and operated on client machines using a standard WWW browser. The experiment content was served by a 200MHz Pentium Pro PC running Microsoft's *Windows NT v4.0* and *Internet Information Server* software. The client machines were 233MHz Pentium II PCs running *Windows NT v4.0* and Netscape's *Communicator v4* web browser, the latter using the *Cosmo Player v2.1* VRML browser to view 3D scenes. In addition, each client machine was running Microsoft's SQL database software in order to store user-specific information during each experiment and also to log the final choice of fitting room made by the participants. The specific details for the individual components of the experiment are provided below.

Virtual Clothing Shop

The virtual clothing shop, in which the experiment participants examined and selected clothing items, was implemented as a VRML scene within an HTML page. A hidden Java applet was also present within the HTML page to enable the most recent selection of clothing item to be registered on the local database.

Mannequin Customisation Interface

The mannequin customisation interface, by which the participants personalised the virtual mannequin to their own dimensions and skin colour, was implemented using a VRML scene (containing the mannequin, height indicator and background objects) linked to a Java applet within an HTML page. The applet contained all of the controls (buttons, drop-down menus, *etc.*) used to adjust the mannequin and also registered the parameters of the customised mannequin on the local database.

Fitting Room Choice

To allow the participants to enter either of the two virtual fitting rooms, an HTML page containing a VRML scene was used. As with the clothing shop page, a hidden Java applet was also present; in this case, to implement the randomised switching of the modelling scenarios associated with each fitting room.

Fitting Rooms

The fitting rooms themselves were implemented as a Java applet embedded within an HTML page which also provided textual instructions for the participants and the 'exit' button. The garment modelling software was implemented as *native* methods within a Java class, *i.e.* class methods which use platform-specific implementations, compiled from C++ code and residing in dynamically linked libraries on the client machine.

The applet itself determined which modelling specification (see previous section) to use according to a script filename specified in the parameters passed to the applet from the HTML page. The script (corresponding to either specification) was downloaded and modified to reflect *(a)* the gender, dimensions and skin colour of the virtual mannequin and *(b)* the size and style of the selected clothing item, the details of both being obtained from the local database. After initializing the modelling software, the applet obtained each image rendered internally by the software, superimposing a number corresponding to the simulation frame before displaying it within the HTML page.

When the execution of the applet was terminated, as a result of the participant clicking the 'exit' button, the details of the modelling scenario used (A or B) and the number of frame at which the participant opted to leave were registered on the local database for later analysis.

## 7.6 Results

The preferences expressed by the experiment participants for the two fitting rooms (according to the simulation specifications) were as follows:

Specification A : **22%** (11/50 users)
Specification B : **78%** (39/50 users)

Of those 11 participants who opted for specification A, the reasons for the choice (provided by the final questionnaire, see Appendix E) were distributed as follows:

| | | |
|---|---|---|
| It was more helpful | **45%** | (5/11) |
| It was more efficient | **27%** | (3/11) |
| The virtual mannequin looked more realistic | **36%** | (4/11) |
| The clothing looked more realistic | **55%** | (6/11) |
| It took less time to dress the mannequin | **27%** | (3/11) |
| It was more enjoyable | **9%** | (1/11) |
| Other reason | **73%** | (8/11) |

Other reasons included:

- that the "clothes hung better"
- that the choice was "random"
- that "they [the two fitting rooms] seemed the same"

Of those 39 participants who opted for specification B, the reasons for the choice (provided by the final questionnaire, see Appendix E) were distributed as follows:

| | | |
|---|---|---|
| It was more helpful | **36%** | (14/39) |
| It was more efficient | **56%** | (22/39) |
| The virtual mannequin looked more realistic | **21%** | (8/39) |

| | | |
|---|---|---|
| The clothing looked more realistic | **26%** | (10/39) |
| It took less time to dress the mannequin | **85%** | (33/39) |
| It was more enjoyable | **38%** | (15/39) |
| Other reason | **31%** | (12/39) |

Other reasons included:

- that the choice was "random"
- that "both fitting rooms were the same"
- that "it was faster in movement"
- that "that was where the wee star landed and ∴ it was quicker"
- that the "blue room [the colour of the other fitting room in this case] took longer to go through dressing"
- that "blue [the colour of the fitting room in this case] is a calming colour, the colour of sky, sea and tranquillity"

The results of the expressed preferences and the associated reasons are shown in graphical format in Fig. 7.8.

The results of the user attitude questionnaires for each fitting room (according to the simulation specifications) are shown in Fig. 7.9. On the vertical scale, a value of 1 corresponds to the least positive attitude, a value of 7 to the most positive, and a value of 4 to a neutral attitude.

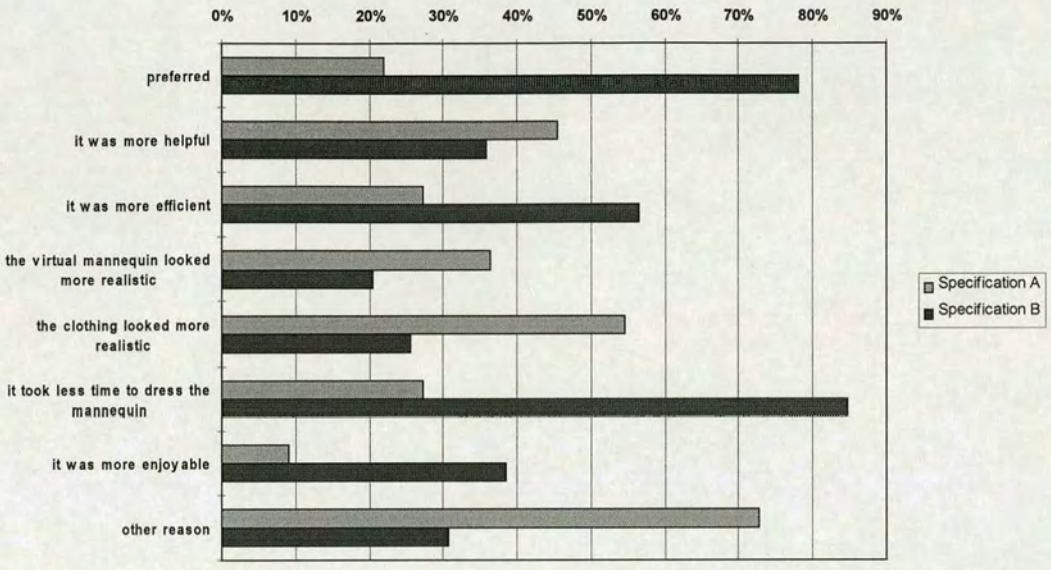



Figure 7.8: Expressed preferences and associated reasons[3]

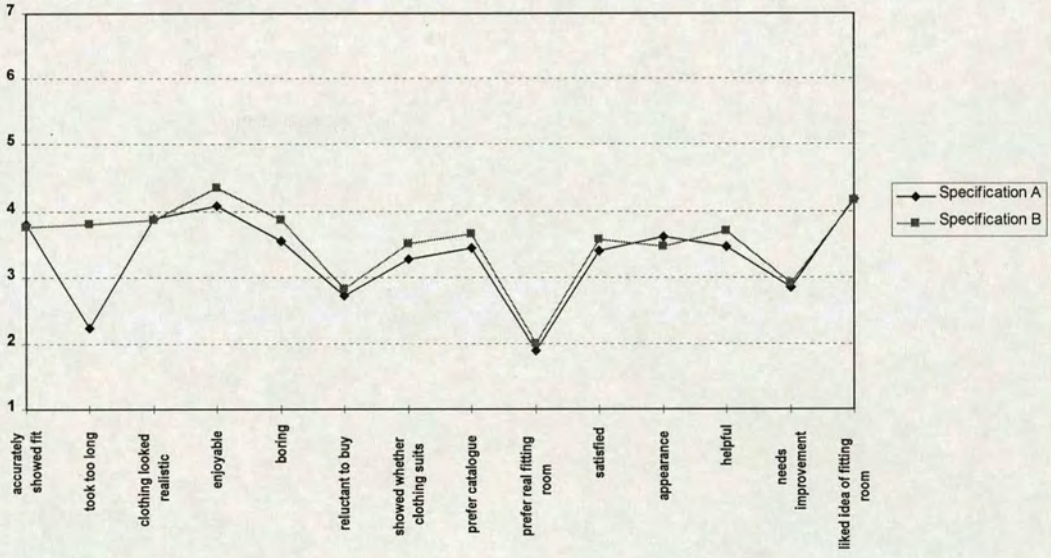



Figure 7.9: Results of user attitude questionnaires

[3] Note: In the percentages shown, only the first category ('preferred') indicates a proportion of the *entire* user sample. The remaining categories indicate the proportion with respect to the number of users who expressed a preference for *that* specification.

## 7.7 Analysis

A significant majority of the experiment participants expressed a preference for the fully FIGMENT-based fitting room service. Those who chose this service did so predominantly because of the increased speed and those who chose otherwise in many cases claimed to perceive no difference between the two. Moreover, the results of the user attitude questionnaires for the two fitting rooms show that no significant loss of accuracy, fidelity, quality or usability was perceived by the participants. Thus, the original hypothesis has been conclusively confirmed.

The experimental results also indicate that a significantly improved mannequin service would be required before customers would feel comfortable and confident making use of it. It is encouraging to note, however, that participants were open to the possibility of using a virtual mannequin service even on the basis of such an early and admittedly deficient implementation.

## 7.8 A VRML-based implementation

VRML[4] is a script-based format for specifying complex, animated and interactive 3D objects and scenes which may be viewed or navigated by any WWW browser with the appropriate enhancements.[5] Using VRML, 3D environments may be 'inhabited' and 'explored', and 3D objects may be examined by viewing from a potentially infinite number of angles and distances. Since VRML has become the internationally recognised standard for 3D modelling via the Internet, it is therefore worth considering whether a VRML-based implementation of a FIGMENT-based mannequin service would be possible and advantageous.

[4] Virtual Reality Modelling Language. See **http://www.vrml.org** and **http://vag.vrml.org** for details of the history and specification of the language.

[5] For example, Microsoft's *Internet Explorer* browser and Netscape's *Navigator* browser may both be enhanced to view VRML scenes by installing SGI's *Cosmo Player* plug-in, Intervista's *WorldView* plug-in, or Sony's *Community Place* plug-in.

There are two potential advantages to a VRML-based mannequin service. Firstly, such a service would be making use of pre-installed and configured 3D software rather than having to supply its own (as was done in the experimental service detailed previously). Secondly, a dressed mannequin modelled in VRML could be interactively viewed from any desired angle within the appropriate browser. With regard to the latter, however, real-time interaction with the VRML scene would not be possible *during* simulation due to the high computational load which the modelling software places upon the client machine. Simulation would need to be stopped or paused in order for the user to interactively view the mannequin; because this would normally be done only after the mannequin has been satisfactorily 'dressed', this causes no significant problem for a VRML-based service.

The first three points of the FIGMENT scheme present no obstacle to a VRML-based implementation, since these are concerned with the manipulation of the internal representations of the mannequin and cloth meshes. Connecting the internal representations and the visible VRML objects may be achieved by using the *VRML External Applications Interface* which allows a Java applet to read and write to the fields of VRML nodes, *e.g.* to update vertex positions. Implementing the fourth point of the FIGMENT scheme—the hybrid rendering algorithm—is clearly less straightforward. Either it must be omitted altogether, allowing for the possibility of unsatisfactory rendering discrepancies, or else a customised VRML browser must be used which implements nodes[6] parallel to the Open Inventor extensions described in Chapter 6.

Producing a customised VRML browser may not be as great a task as it initially appears; the source code for a number of already-existing browsers has been made available by their developers. Source code for VRML parsers and 3D rendering libraries is also freely available. However, requiring the user to have a customised browser defeats the first of the two potential advantages of a VRML-based implementation offered above. On further reflection it appears that most of the full functionality of a VRML browser is unnecessary in order to obtain the second

[6] VRML was originally based on Open Inventor and is similar in structure, using hierarchical scene graphs containing various nodes which describe the appearance and geometry of 3D objects.

advantage, *i.e.* the ability to view a dressed mannequin from various angles. It would be relatively straightforward to develop the Java/Open Inventor solution used in the experiment described above to allow for the same feature. Extra buttons and controls could be added to the Java applet which displays the rendered mannequin to enable the user to rotate the scene and to 'zoom in and zoom out' in a similar way to a VRML browser. Furthermore, since the user would still require additional software for a VRML-based mannequin service, there is no great disadvantage in supplementing that software to provide interactive 3D viewing of the mannequin.

In conclusion: despite the potential advantages of a VRML-based mannequin service, the details of the FIGMENT scheme would provide problems for a full implementation, while the same benefits could be obtained by a non-VRML-based service, *e.g.* a Java applet with native methods which uses the Open Inventor graphics library.

## 7.9 Conclusions

The results of the experiment described in this chapter provide substantial support for affirming that the FIGMENT scheme achieves the goals for which it is intended: to allow the implementation of a virtual mannequin service on a low-end client platform which provides results at interactive speeds yet with no discernible detriment to the accuracy or fidelity of the visual results when compared with a service based on alternative approaches to garment modelling. In the trial, the overwhelming majority of users preferred the fully FIGMENT-based service, primarily on the grounds of speed, while expressing no significant difference in attitude with respect to other aspects of usability. Clearly, a substantial amount of further development would be required before the simple implementation of a mannequin service used in this case could be offered as a satisfactory commercial service, particularly with respect to the accuracy of the mannequin (both in terms of shape and appearance) used for modelling. The currently ambivalent attitude expressed by the participants in this trial to the *general idea* of a virtual fitting room should, in fact, provide encouragement for further improvement as the realities and

possibilities of telepresence shopping become progressively more comfortable in the minds of the general public.

## 7.10 Summary

In this chapter the focus was shifted from considering the theory behind the FIGMENT scheme and independent analysis of its constituents to its full implementation in practice within a prototypical virtual mannequin service. After providing timing results for two typical modelling scenes to demonstrate the overall speed gains possible by implementing the FIGMENT scheme, the chapter described a user experiment performed using a prototype FIGMENT-based mannequin service. The experiment successfully established, by both objective and subjective measures, that the goal of the FIGMENT scheme has been achieved; it also provided further direction for the commercial development of such a service. Finally, a VRML-based implementation of a FIGMENT-based mannequin service was considered and found to be problematic in certain details while failing to provide any overall benefits which could not be achieved through simpler alternative means.

# Chapter 8

# Conclusions

## 8.1 Introduction

Chapters 2 through 6 have presented the four points of the FIGMENT scheme, detailing the calculations and algorithms necessary for implementing the scheme and also providing empirical data regarding the speed increases afforded and the corresponding accuracy costs incurred by each aspect of the scheme. Chapter 7 moved from theory to practice by describing a user trial aimed to establish the suitability of the FIGMENT scheme for an implementation of a virtual mannequin service. The possibility of an implementation which utilises the VRML 3D scene description language and corresponding software was also discussed.

In this final chapter of this thesis, the FIGMENT scheme is assessed as an integrated entity, summarizing the individual and collective advantages of its components and noting their mutually supportive roles. The limitations of the scheme in its present form are discussed while suggesting profitable avenues for future work. In conclusion, the FIGMENT scheme is offered as a unique and workable solution to the problems of implementing an interactive garment modelling service which provides usable and informative results on the appropriate client hardware platforms, both now and for the imminent future.

## 8.2 Assessment

The experimental results provided in Chapters 2 through 6 establish quite satisfactorily the speed advantages offered by each individual point of the FIGMENT scheme, while also supporting the contention that the computational inaccuracies introduced in each case do not affect the fidelity of the visual results to the extent that the usability and informativeness of a FIGMENT-based mannequin service would be hindered.

The physical model provides a robust foundation for the scheme, allowing flexibility in the garment models used for simulations. The instability-countering measures incorporated into the model are computationally efficient and allow for a considerable increase in modelling rates with a negligible difference in results. For particularly time-critical implementations, the faster methods for estimating tensional and flexional forces permit an even greater speed increase.

The second point of the scheme—collision volume approximation—proves to be an excellent compromise between traditional real-time and non-real-time collision detection algorithms. The 'capsule' method provides exceptionally fast collision handling, while the 'radial depth' method provides a considerably more accurate representation of the mannequin body with little additional computation. The methods may be combined for optimum performance in any particular application and both demonstrate a substantial speed advantage over a standard hierarchically-optimised polygon-to-polygon algorithm. Although a significant amount of precomputation is required when obtaining a 'capsule' or 'radial depth' representation of any particular mannequin, the nature of the structure allows instantaneous resizing and animation without further precomputation. Other advantages include the straightforward handling of deep penetrations by cloth mesh nodes (allowing garments to be superimposed over the mannequin rather than being drawn together from a distance) and the compact internal representation of the collision objects.

The use of garment models constructed from 'progressive' meshes also allows a further significant reduction in simulation times by using less complex

representations of cloth meshes during the earlier stages and restoring those meshes in a gradual and practically seamless manner to their original high-resolution form. A once-for-all simplification of a particular cloth mesh allows for considerable flexibility when using the mesh for modelling when specifying both the initial and final levels of complexity and the rate of progression during simulation. As detailed in the relevant chapter, the FIGMENT scheme provides effective algorithms for mesh decimation and reconstruction which, most importantly, take into account the specific problems posed by using progressive meshes for dynamically deformed cloth surfaces rather than static, rigid models.

The hybrid rendering algorithm, concluding the four points of the FIGMENT scheme, avoids the deficiencies of both depth-buffered and depth-sorted rendering algorithms by combining features of both and allowing cloth surfaces to be adequately rendered without resorting to computationally costly algorithms to maintain precise geometric accuracy. The algorithm is particularly effective in its ability to render complex, folded, multiple-mesh garments (and in multiple overlapping layers) without introducing glaring visual discrepancies. All this is achieved with minimal increase in rendering times.

Having summarised the individual advantages of the four points of the scheme, it is important to also note the various supportive relationships between the points.

Firstly, the use of progressive meshes would not be possible without a physical model which is based on *(1)* finite elements, *(2)* triangular cloth sections, and *(3)* irregular meshes.

Secondly, as well as allowing for significantly larger simulation time steps, the instability-countering measures also effectively suppress the transient distortions in the mesh introduced by *(a)* the correction of deep penetrations of the collision volume approximation objects and *(b)* the reconstruction of progressive meshes. Without these measures, the second and third points of the FIGMENT scheme would be practically unusable; moreover, these transient deformations are dealt with in a manner which avoids compromising the fidelity of the results.

A third supportive relationship occurs between the collision volume approximation methods and the use of progressive meshes. During the initial frames

of simulation, the sections of the cloth mesh are relatively large and would require a polygon-to-polygon collision detection algorithm to 'work harder' in order to correct penetrations and to avoid the cloth mesh actually 'slipping' through the surface of the mannequin.[1] In contrast, the capsule and radial depth methods of collision detection are much more robust in handling low-complexity meshes and require no additional computation in such cases.

Fourthly, while the mannequin is being 'dressed', the hybrid rendering algorithm effectively shields the viewer of the modelling scene from the minor inaccuracies involved in *(a)* approximating the surface of the mannequin for collision purposes (*i.e.* in some regions the mannequin will inevitably protrude to a marginal extent from its collision object representation) and *(b)* the inability of lower-complexity meshes to closely follow the surface of the mannequin (*i.e.* in the initial stages of mesh progression, although large sections may in actual fact penetrate the mannequin at their central regions (Fig. 8.1), they appear to not do so).

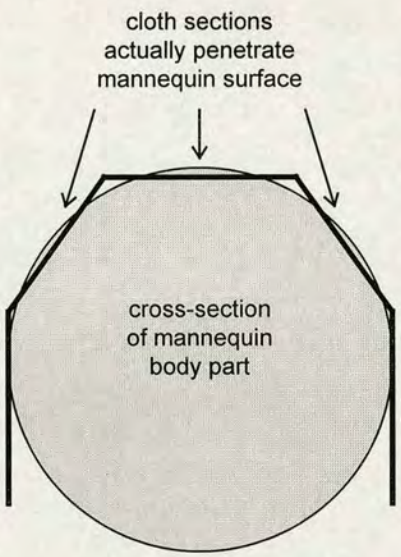



Figure 8.1: Section penetration in low-complexity meshes

Thus, it can be seen that not only does each point of the FIGMENT scheme contribute individually toward the reduction of modelling times, but that in a number

[1] Consider the role of the 'proximity' parameter required in the implementation of the polygon-based algorithm detailed in Appendix A.

of significant ways each point relies on other features of the scheme for its effectiveness or even its very possibility.

## 8.3 Future work

At this point, the various limitations of the FIGMENT scheme as it has been presented here should be summarised and considered in order to suggest directions for future work and development.

With regard to the physical model, it will be noted from the visual results of the example simulations provided in Chapter 2 that the use of the *faster* methods for estimating tensional and flexional internal forces within cloth meshes lead to marginally different results than when using the *standard* methods, particularly in cases which feature loose, folded portions of cloth (*e.g.* dresses). Although this discrepancy may be expected, and ought not to greatly affect the usability of a mannequin service which employs the former methods, a fruitful line of inquiry would involve trying to develop a computationally efficient method of compensating for the error introduced. A solution would presumably focus on the difference between the flexional forces computed in each case and might only involve applying a compensating factor of the form $k\theta^n$, $k\sin^n\theta$ or $k\cos^n\theta$ to the bending forces acting on the nodes surrounding a joint (where $\theta$ is the angle between the sections forming the joint, $k$ is a constant and $n \in \{0,1,2,\ldots\}$). Further improvement of the physical model might involve applying a more sophisticated integrative method when updating the dynamics of the mesh during each iteration of simulation, *e.g.* a Runge-Kutta method, although the computational cost incurred may not merit the resultant gain in accuracy.

Although the 'radial depth' method of collision volume approximation can provide a remarkably close representation of the mannequin body surface, there is room for improvement in the 'capsule' structure to allow a more accurate 'fit'. One potentially effective approach might involve using a more complex tapering function—polynomial or sinusoidal, perhaps, rather than linear—to specify the shape of a capsule object. This function, if thoughtfully chosen, could even allow for the

removal of the two rounding factors and thus reduce the present three-part structure of the capsule to a single element. The development of alternative collision volume approximation structures besides the two offered here should also be considered.

There are a number of avenues for improvement in the use of progressive meshes within the scheme. At present, implementations of the decimation algorithm for garment models can take a considerable amount of time to execute; the method described here considers every valid edge-collapse transformation for a mesh by evaluating the energy function for the resultant mesh. It would therefore be worthwhile to investigate the possibilities of speeding up the process by heuristically narrowing the set of potential edge-collapse transformations for a particular mesh through the elimination of 'obviously' unsuitable candidates. In addition, it will be noted that no attention is currently paid to the effect of increased mesh discretisation on the *physical* characteristics of the cloth—specifically, the compensation factor used in the calculation of bending forces acting at the joints of cloth sections (see Section 2.3). A more precise implementation would increase this compensation factor accordingly with respect to the reduction in mesh complexity, decreasing the factor as mesh reconstruction proceeded. However, the accuracy analysis provided in Chapter 5 indicates that the current absence of this consideration may have little effect on the final drape of the garments.

Finally, Section 6.5 has already discussed several limitations of the implementation of the FIGMENT hybrid rendering algorithm detailed in that chapter. To reiterate: *(1)* certain poses and views of the mannequin in modelling scenes cannot be satisfactorily handled by the implementation of the *StencilClear* and *StencilSeparator* nodes offered here, and *(2)* the absence of cloth-to-cloth collision handling permits cases of inner layers of garments appearing to protrude from behind outer layers. The implementation of the FIGMENT scheme in a virtual mannequin service would therefore profit from the development of solutions along the lines of those suggested in Chapter 6.

## 8.4 Conclusions

In the first chapter of the present work, the difficulties of implementing a comprehensive 'telepresence' shopping environment using currently available technology were introduced and the concept of a 'virtual mannequin' offered as a viable and versatile solution. The practical obstacles to the realisation of such a service using previously developed modelling and rendering techniques were also discussed, concluding that a fairly significant change in perspective would be required in order to meet the conflicting goals of *(a)* fast modelling rates and *(b)* faithful and informative results, both of which are necessary for a truly usable application. A solution which meets both requirements at an optimal point of compromise is needed if such a service is to be a reality in the near future, rather than simply waiting for the next generation (or two) of domestic computer hardware to roll off the production line.

It is the contention of this concluding section that the FIGMENT scheme, described and detailed in the preceding chapters, provides such a solution. The four points of the scheme, when combined and implemented, allow the dressing of a detailed mannequin model with multiple complex garments in a matter of minutes on a mid-level desktop PC platform, providing the user with an informative view of prospective clothing purchases as an invaluable supplement to alternative visual media. Furthermore, once the dynamic modelling is complete, the user may view the mannequin from a number of angles and instantly change the colour, texture, and motifs of clothing items without recomputation. The 'trying on' process may be halted at any point once the user has gained enough information about his or her garment selections; a different selection of sizes, styles, and combinations may then be viewed. Although such a mannequin service cannot provide the 'instant' results to which modern consumerism is becoming accustomed, it can still match the time typically required to dress oneself in a real fitting room. In fact, this period can provide the opportunity for further browsing of items, other shopping, entertainment and commercials—none of which will be running against the interests of those providing the service. It is no understatement to suggest that the FIGMENT scheme

could make possible precisely that which the fashion industry needs in order to gain a respectable and profitable presence in the world of electronic retailing.

The speed gains contributed by each point of the scheme to the simulation of typical modelling scenes have been shown to be substantial. Furthermore, empirical accuracy analysis, subjective assessment of visual results, and the outcome of the user trial detailed in Chapter 7, all support the conclusion that the fidelity and usability of a FIGMENT-based mannequin service would not be adversely affected by the used of such time-reducing algorithms and techniques. Thus, FIGMENT achieves the goal for which it is intended.

Finally, although the various methods employed by the scheme have been developed with a specific application in mind, techniques such as collision volume approximation may well prove to be profitable in other scenarios, *e.g.* interaction between complex avatars and other objects in multi-user environments.

# Appendix A

# Polygon-Based Collision Algorithm

This appendix provides details of the polygon-based collision handling algorithm used for comparison against the volume approximation methods presented in Chapters 3 and 4. As with those methods, the algorithm consists of two parts: the collision detection method and the collision response method.

Collision detection

The basic algorithm employed is that presented in Yang (1993) which consists of an efficient hierarchical octree-subdivision algorithm with $O(n \log n)$ complexity. The use of this particular algorithm for comparison purposes is particularly appropriate due to its application, in the context of Yang (1993), for cloth modelling. The function of the algorithm is to detect whether a triangular face intersects with one or more members of a set of $n$ triangular faces. If intersections are detected, then the collision response algorithm is applied to that face.

For extended details of the detection algorithm, the reader is referred to the original paper. However, a brief summary of the algorithm is provided here.

A set of triangular faces is compiled with which intersections will be tested, *e.g.* the set of polygons comprising the (significant) mannequin body parts, and the bounding box for each face is computed. A randomly-generated index is chosen for each face (this ensures a generally well-balanced tree structure, see below) and the set of faces is arranged into an ordered list according to the index of each face.

The first face in the resultant list is taken as the root node of a hierarchical 'octree' structure in which each node is associated with up to eight child nodes. Each of the eight sub-trees corresponds to a *sub-space* in three-dimensional space (Fig. A.1) with respect to the *minimum* corner of the bounding box of the parent node (*i.e.* the associated triangular face). As each face is taken from the ordered list, the minimum corner vertex (*i.e.* with respect to its $x$, $y$ and $z$ coordinates) of its bounding box is compared with the minimum corner vertex of the bounding box of the *root* node, to determine within which of the eight sub-spaces that vertex lies. If the root node currently has no child node corresponding to that sub-space, then the face taken from the list becomes that child node; otherwise, the face is compared with the *child* node in the same way, and so forth until a place is available for the face within the octree. The process continues with the *next* face in the ordered list until all have been placed into the octree. The algorithm for creating the octree of faces has $O(n \log n)$ complexity.

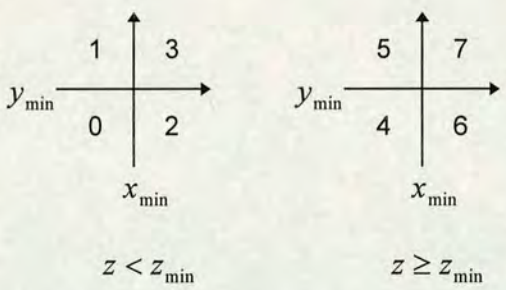



Figure A.1: Minimum corner vertex of bounding box dividing space into 8 sub-spaces

The algorithm for detecting whether an arbitrary triangular face **f** intersects with any of the faces within the octree proceeds in the following manner. The maximum and minimum corner vertices of the bounding box of **f** are compared with the minimum corner vertex of the bounding box of the *root* node (face) of the octree to determine in which of the eight sub-spaces they lie. With this information, certain of the eight sub-trees of the root node may be eliminated from the detection, since the bounding boxes of the faces contained in those sub-trees *cannot possibly* intersect with that of **f** (see Yang (1993) for details of this simple process). For the remaining sub-trees, **f** must then be compared with the minimum corner vertex of the root node

of each sub-tree (*i.e.* each child node of the topmost root node) in the same manner, and so forth descending throughout the entire octree. Additionally, if one of those remaining sub-trees corresponds to the sub-space in which the face of the *parent* node lies, **f** must be tested for intersection with that face itself. Thus, the algorithm may be expressed as a recursive function in pseudo-code:

```
function collides( Face f, Node root )
{
   Box box = root.boundingBox;
   foreach ( sub-space of box.minimumCornerVertex )
   {
      Node child = root.child[sub-space];
      if ( possiblyIntersectsSubtreeFaces( f, child ) )
         collides( f, child );

      if ( containsBox( sub-space, box ) )
         intersects( f, root.face );
   }
}
```

The algorithm for checking for intersection between a pair of triangular faces proceeds thus. First, a check for intersection between the bounding boxes of the faces is performed; if the bounding boxes intersect, then each of the three edges (line segments) of one face is tested for intersection with the other face. The complexity for the intersection-detecting algorithm is also $O(n \log n)$ and thus the complexity of the overall algorithm is $O(n \log n)$. However, although the intersection-detecting algorithm must be executed for each iteration of the physical simulation, the octree-creating algorithm need only be executed at the beginning of the simulation unless the geometry of the mannequin body changes during the simulation (as it must do if the mannequin is to be animated).

Collision response

Each triangular section (face) of a cloth mesh must be tested against the octree of triangular faces comprising the mannequin body. For each intersection detected (if any) a collision response must be applied to those nodes (vertices) of the mesh which have penetrate the body. This response involves correcting the position of the node

and applying the appropriate change in the dynamics of the node, a required the computation of a corrected position (*i.e.* the closet position on the surface of the body) and the normal vector at that point (cf. Sections 3.8 and 4.4). Also required is the coefficient of friction at that point.

Thus, when an intersection is detected between a section of a cloth mesh and a face $\mathbf{f}$ from the mannequin body according to the detection algorithm described above, it must be determined which nodes of that section have penetrated the body. This is done by simply calculating, for each edge of the cloth section which intersects $\mathbf{f}$, which of the two nodes comprising that edge is on which side of the *plane* of $\mathbf{f}$. Having established which node requires correction, the closest point on the surface of the mannequin body must be determined. If it were the case that *(a)* the sections of the cloth mesh were substantially smaller in size than the polygons of the mannequin body and *(b)* the penetrations of the nodes of the mesh were of relatively little depth, then it would be sufficient to calculate the closest point on the one polygon with which intersection has been detected. However, in the comparison simulations run in order to evaluate the FIGMENT collision methods, neither of these conditions hold. For this reason, both that one polygon *and those neighbouring polygons lying within a specified proximity* must be considered when determining the closest surface point to the penetrating node. Thus, a 'proximity' parameter must be specified for each simulation to indicate the range of neighbouring polygons which will be checked to determine the closest surface points of penetrating nodes and hence ensure a smooth and accurate collision response. Once a surface point has been established, the normal vector and frictional coefficient are simply taken as those of the polygon on which this point occurs.

One final issue required attention in the implementation of the collision handling algorithm. When a node of the cloth mesh is found to have penetrated the mannequin body, this will normally be the result of *more* than one intersection between faces of the mesh and the body (Fig. A.2). For this reason, for each intersection detected, the collision response information (*i.e.* corrected surface point, surface normal vector and frictional coefficient) computed for each penetrating node is *stored* until the end of the detection process and the response information with the corrected surface point

*closest* to that penetrating node is used in the final calculation of the collision response for each affected node. Having determined which nodes of the mesh have penetrated the mannequin body, the collision response is applied to those nodes in the same way as for the capsule and radial depth approximation methods of the FIGMENT scheme.

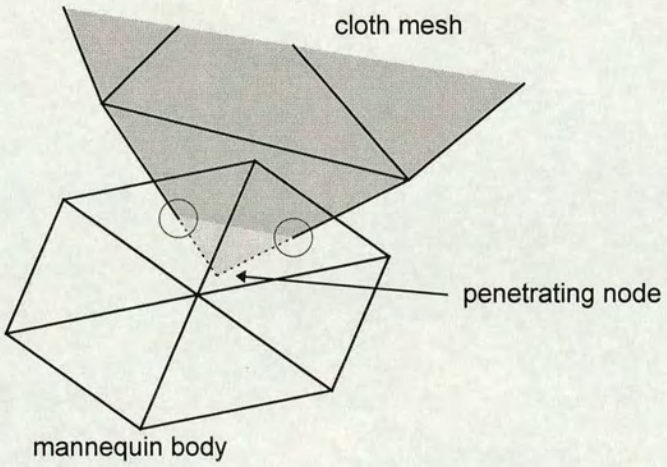



Figure A.2: Penetration of mannequin body by node detected by two instances of intersection between faces of mesh and body

# Appendix B

# Normal Computation for Radial Depth

This appendix provides a set of calculations which would allow for a more precise determination of the surface normal vector for a mesh node which penetrates a radial depth collision object (see Section 4.4). The following calculations all apply to a radial depth object with its lateral axis aligned to the vertical axis, thus the normal vector obtained from the calculations will require appropriate transformation into global space depending on the orientation of the object itself.

A straightforward way to calculate the surface normal vector on the surface of a three-dimensional object is to compute the cross-product of two unit vectors which represent tangents to the surface in the horizontal and vertical planes (Fig. B.1). The first vector, $\overline{u}_H$, is aligned at a tangent to a horizontal cross-section (Fig. B.2); the second, $\overline{u}_V$, aligned at a tangent perpendicular to that cross-section.

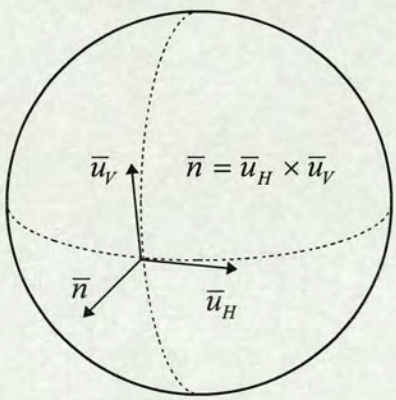



Figure B.1: Calculation of surface normal from tangential vectors

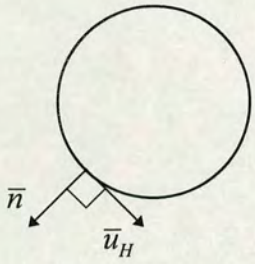



Figure B.2: First tangential vector taken from horizontal cross-section

For a radial depth object, then, the second tangential vector, $\overline{u}_V$, may be calculated from *(a)* the slope between the two cross-sections which surround the surface point in question and *(b)* the bearing $\theta$ of the surface point around the vertical axis. Thus, referring to Fig. B.3, the tangential unit vector $\overline{u}_V$ is computed to be

$$\overline{u}_V = \frac{1}{\sqrt{\Delta d^2 + h^2}}\left(\Delta d \cos\theta, h, \Delta d \sin\theta\right) \tag{B.1}$$

where

$$\Delta d = d_1 - d_2$$

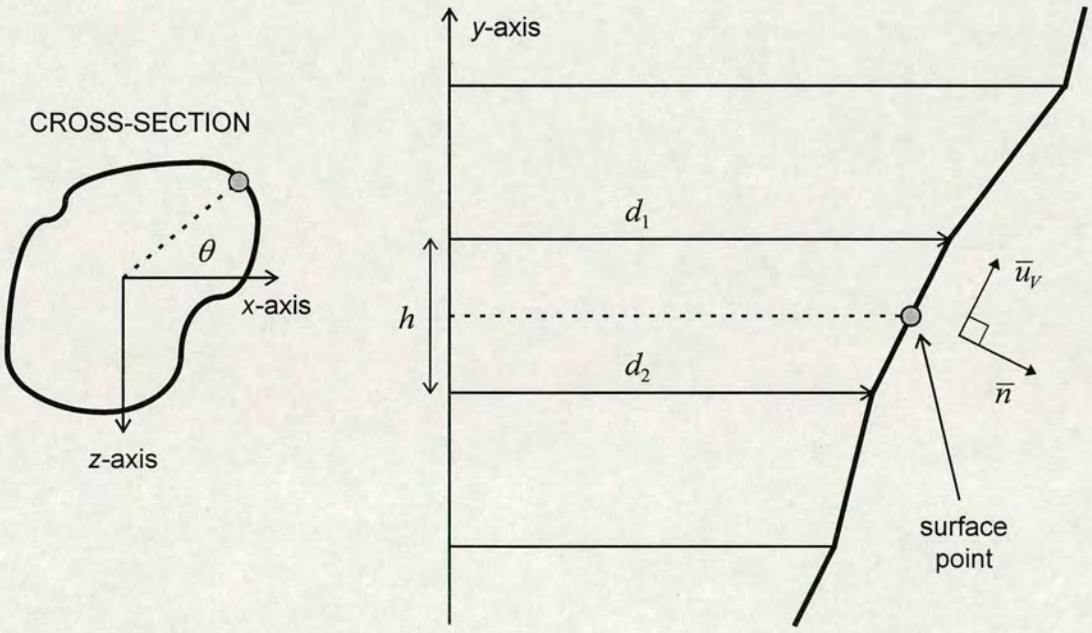



Figure B.3: Second tangential vector calculated from slope between cross-sections

Computing the first tangential vector, $\overline{u}_H$, is slightly more involved since it must take into account the gradient, *i.e.* the derivative, of the radial depth function which defines the cross-section of the object at the surface point. In general, that radial depth function must be obtained by interpolating the two radial depth functions corresponding to the two intervallic cross-sections which surround the surface point (as in Fig. B.3). Thus, for the resultant Fourier-approximated radial depth function,

$$D_{approx}(\theta) = A_0 + \sum_{n=1}^{M} A_n \cos n\theta + \sum_{n=1}^{M} B_n \sin n\theta \tag{B.2}$$

the gradient at any particular point is given by the derivative function,

$$D'_{approx}(\theta) = -\sum_{n=1}^{M} A_n \sin n\theta + \sum_{n=1}^{M} B_n \cos n\theta \tag{B.3}$$

where $\{A_0, A_1, \ldots A_M, B_1, \ldots B_M\}$ are the (interpolated) Fourier coefficients corresponding to the cross-section at the surface point.

Having obtained the gradient $D'_{approx}(\theta)$ of the radial depth function at the surface point in question, it is relatively straightforward to compute a value for the vector $\overline{u}_H$. Consider the example cross-section shown in Fig. B.4(a) and the corresponding radial depth function shown in Fig. B.4(b); furthermore, consider both to be mapped onto the same two-dimensional coordinate system (corresponding to the *x*-*z* plane). With some reflection, it can be seen that a two-dimensional unit vector which is tangential to the *radial depth function* at point *A*, *i.e.* (0.707,0.707), if reflected in the *z*-axis and then rotated $90^\circ$ anticlockwise around the origin corresponds to a two-dimensional unit vector which is tangential to the *cross-section* at point *A*, *i.e.* (0.707,0.707). The same applies to unit vectors tangential at points *B* and *C*, *i.e.* (1,0) and (0.707,-0.707), if reflected in the *z*-axis and then rotated $45^\circ$ and $0^\circ$ anticlockwise around the origin, respectively, *i.e.* (-0.707,0.707) and (-0.707,-0.707).

Hence, to generalise for the surface point at bearing $\theta$, the two-dimensional unit vector $\overline{u}$ which is tangential to the *radial depth function* at that value of $\theta$ is given by

$$\overline{u} = \left(u_x, u_z\right) \tag{B.4}$$

where

$$u_x = \frac{1}{\sqrt{1 + D'_{approx}(\theta)^2}} \text{ and } u_z = \frac{D'_{approx}(\theta)}{\sqrt{1 + D'_{approx}(\theta)^2}}$$

and the corresponding two-dimensional unit vector $\overline{u}'$ tangential to the *cross-section* at that value of $\theta$ is given by

$$\overline{u}' = \left(-u_x \cos\phi - u_z \sin\phi, -u_x \sin\phi + u_z \cos\phi\right) \tag{B.5}$$

where

$$\phi = \frac{\pi}{2} - \theta$$

*i.e.* $\overline{u}'$ is the result of reflecting $\overline{u}$ in the $z$-axis and then rotating by an angle of $(90° - \theta)$ anticlockwise around the origin. Having obtained $\overline{u}'$, $\overline{u}_H$ is simply the corresponding unit vector in three-dimensional space:

$$\overline{u}_H = \left(-u_x \cos\phi - u_z \sin\phi, 0, -u_x \sin\phi + u_z \cos\phi\right) \tag{B.6}$$

where

$$u_x = \frac{1}{\sqrt{1 + D'_{approx}(\theta)^2}} \text{ and } u_z = \frac{D'_{approx}(\theta)}{\sqrt{1 + D'_{approx}(\theta)^2}} \text{ and } \phi = \frac{\pi}{2} - \theta$$

As explained above, the surface normal vector is finally computed to be the vector cross-product of unit vectors $\overline{u}_H$ and $\overline{u}_V$. Care should be taken to ensure that the normal vector is correctly computed to be directed away from the surface of the radial depth object, rather than toward it.

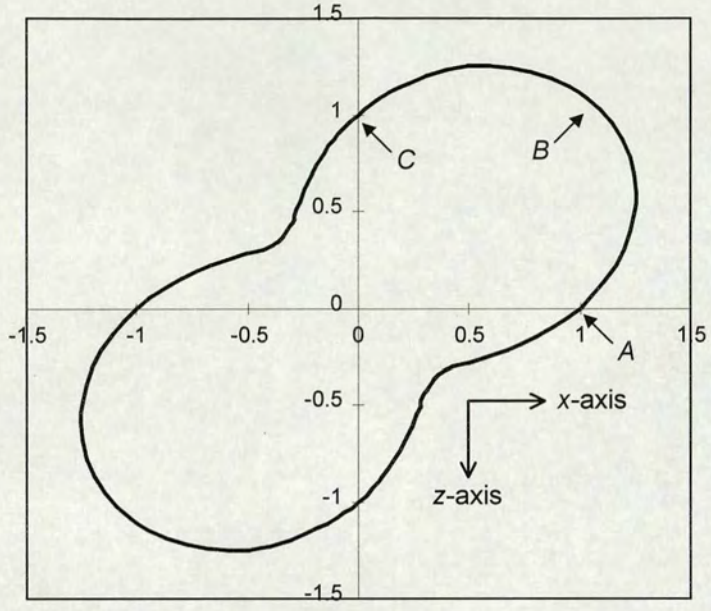



Figure B.4(a): Example cross-section of radial depth object

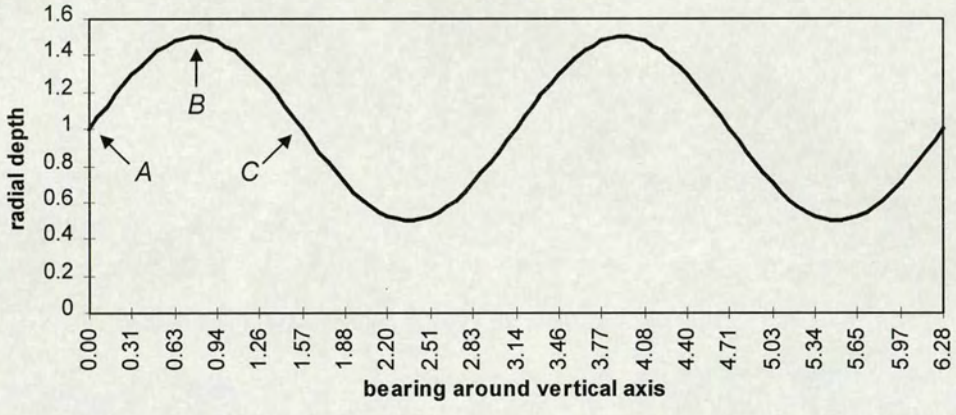



Figure B.4(b): Radial depth function for example cross-section

# Appendix C

# Open Inventor Extensions

This appendix provides the C++ code used to implement the *ClothingLayer*, *DepthSortedGroup*, *StencilClear* and *StencilSeparator* nodes as extensions to the Open Inventor 3D graphics toolkit. For full details of how extensions to the Inventor library are implemented, see Wernecke (1994b). For details of the OpenGL functions used in the following code, see Woo, Neider and Davis (1997); also Kempf and Frazier (1997).

FILE: ClothingLayer.h

```
/*****************************************************************************/
/* ClothingLayer class header.                                               */
/*                                                                           */
/* Extends IndexedFaceSet to implement FIGMENT hybrid rendering algorithm */
/*                                                                           */
/* James Anderson (c) 1995-1998                                              */
/*****************************************************************************/


#ifndef _CLOTHINGLAYER_H_
#define _CLOTHINGLAYER_H_


#include <Inventor/nodes/SoIndexedFaceSet.h>
#include <Inventor/fields/SoSFBool.h>
#include <Inventor/fields/SoSFFloat.h>
#include <Inventor/fields/SoMFFloat.h>
#include <Inventor/fields/SoMFInt32.h>
#include <Inventor/actions/SoGLRenderAction.h>


#ifdef WIN32
#include <SoWinLeaveScope.h>
#endif


// Class definition
//
class ClothingLayer : public SoIndexedFaceSet
{

  SO_NODE_HEADER( ClothingLayer );
```

```
public:

  // Fields
  SoSFBool doOverlap;
  SoSFBool doDepthSort;
  SoSFFloat criticalCosineAngle;
  SoSFBool doStencil;

  // Initializes the class
  static void initClass();

  // Constructor
  ClothingLayer();

protected:

  // Method to render object
  virtual void GLRender( SoGLRenderAction *action );

private:

  // Destructor
  virtual ~ClothingLayer();

  // Method to determine whether triangle is facing toward or away from viewpoint
  SbBool isFacing( const SbVec3f& p0, const SbVec3f& p1, const SbVec3f& p2,
                   const SbMatrix& matrix, float cosine );

  // Function to quicksort array of indices according to an associated value
  void quicksort( int *index, float *value, int a, int b );

  // Arrays used when ordering faces
  float *faceDistance;
  int *faceIndex;
};


#endif // _CLOTHINGLAYER_H_
```

FILE: ClothingLayer.cpp

```
/*****************************************************************************/
/* ClothingLayer class body.                                                 */
/*                                                                           */
/* Extends IndexedFaceSet to implement FIGMENT hybrid rendering algorithm */
/*                                                                           */
/* James Anderson (c) 1995-1998                                              */
/*****************************************************************************/


#include <Inventor/misc/SoState.h>
#include <Inventor/elements/SoCoordinateElement.h>
#include <Inventor/elements/SoTextureCoordinateElement.h>
#include <Inventor/elements/SoTextureCoordinateBindingElement.h>
#include <Inventor/elements/SoGLTextureEnabledElement.h>
#include <Inventor/elements/SoGLLazyElement.h>
#include <Inventor/elements/SoLazyElement.h>
#include <Inventor/elements/SoMaterialBindingElement.h>
#include <Inventor/elements/SoModelMatrixElement.h>
#include <Inventor/elements/SoViewingMatrixElement.h>
#include <Inventor/bundles/SoNormalBundle.h>

#include "ClothingLayer.h"


SO_NODE_SOURCE( ClothingLayer );


// Static method to initialize class
//
void ClothingLayer::initClass()
{
  SO_NODE_INIT_CLASS( ClothingLayer, SoIndexedFaceSet, "IndexedFaceSet" );
}
```

```
// Constructor
//
ClothingLayer::ClothingLayer()
{
  SO_NODE_CONSTRUCTOR( ClothingLayer );

  SO_NODE_ADD_FIELD( doOverlap, (TRUE) );
  SO_NODE_ADD_FIELD( doDepthSort, (TRUE) );
  SO_NODE_ADD_FIELD( doStencil, (FALSE) );
  SO_NODE_ADD_FIELD( criticalCosineAngle, (0) );
}


// Destructor
//
ClothingLayer::~ClothingLayer()
{
}


// Method to render object using hybrid rendering algorithm based on OpenGL
//
// NOTES:
//
// Normals are recalculated *every* time from the polygons themselves, i.e.
// any bound normals will be ignored.  The only material bindings supported are
// OVERALL, PER_FACE and PER_FACE_INDEXED.  The materialIndex *must* be specified
// in the last case.  The texture coordinate binding is correctly implemented,
// although coordinate generating functions are ignored.
//
// All facets must be triangular or else function will crash.
//
void ClothingLayer::GLRender( SoGLRenderAction *action )
{
  int i, j;

  // Get state from the action
  SoState *state = action->getState();

  // Check that object should be rendered
  if ( !shouldGLRender( action ) )
    return;

  // Get current viewing matrix
  SbMatrix vmatrix = SoViewingMatrixElement::get( state );

  // Determine whether to do textures
  SbBool doTextures =
    ( SoGLTextureEnabledElement::get( state ) &&
      SoTextureCoordinateElement::getType( state )
        != SoTextureCoordinateElement::FUNCTION );

  // Determine whether to send normals
  SbBool sendNormals =
    ( SoLazyElement::getLightModel( state ) != SoLazyElement::BASE_COLOR );

  // Get pointer to coordinate element
  const SoCoordinateElement *coord = SoCoordinateElement::getInstance( state );

  // Get pointer to texture coordinate element
  const SoTextureCoordinateElement *tcoord =
    SoTextureCoordinateElement::getInstance( state );

  // Determine texture binding
  SoTextureCoordinateBindingElement::Binding texbind =
    SoTextureCoordinateBindingElement::get( state );

  // Check whether texture coordinate indices are available
  // (if not, the coordinate index is used instead)
  SbBool texUseCoordIndex = textureCoordIndex.isDefault();

  // Determine material binding
  SoMaterialBindingElement::Binding matbind = SoMaterialBindingElement::get( state );

  // Compute normals (always) using NormalBundle object
  SoNormalBundle nb( action, TRUE );
  nb.initGenerator( coordIndex.getNum() );
```

```
// Generate normals for polygons
nb.beginPolygon();
for ( i = 0; i < coordIndex.getNum(); i++ )
  {
    int index = coordIndex[i];

    if ( index < 0 )
      {
        nb.endPolygon();
        nb.beginPolygon();
      }
    else
      nb.polygonVertex( coord->get3(index) );
  }
nb.endPolygon();
nb.generate( 0, FALSE );
const SbVec3f *normals = nb.getGeneratedNormals();

// Set up OpenGL material state
SoGLLazyElement::sendAllMaterial( state );
SoGLLazyElement *lazyElt = (SoGLLazyElement *) SoLazyElement::getInstance( state );

// Compute overall matrix acting on mesh w.r.t. viewpoint
SbMatrix matrix = SoModelMatrixElement::get( state ) * vmatrix;

// Get value of fields
SbBool overlap = doOverlap.getValue();
SbBool depthsort = doDepthSort.getValue();

// Set up stencil operations accordingly
if ( doStencil.getValue() )
  glStencilFunc( GL_NOTEQUAL, 1, 1 );
else
  glStencilFunc( GL_ALWAYS, 1, 1 );
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
glEnable( GL_STENCIL_TEST );

// Determine order in which to render faces if required
int faceNum = ( coordIndex.getNum() + 1 ) >> 2;
if ( depthsort )
  {
    // Allocate arrays for face indices and corresponding distances from viewpoint
    faceIndex = new int[faceNum];
    faceDistance = new float[faceNum];

    for ( int index = 0; index < faceNum; index++ )
      {
        i = index << 2;

        // Compute centre point of face
        SbVec3f centre = ( coord->get3( coordIndex[i] ) +
                           coord->get3( coordIndex[i+1] ) +
                           coord->get3( coordIndex[i+2] ) ) / 3.0;

        // Transform to point relative to view
        matrix.multVecMatrix( centre, centre );

        float dist = centre.length();

        faceIndex[index] = index;
        faceDistance[index] = dist;
      }

    // Sort array into order of decreasing distance
    quicksort( faceIndex, faceDistance, 0, faceNum-1 );
  }

// Draw each polygon in turn
float cosine = criticalCosineAngle.getValue();
for ( int face = 0; face < faceNum; face++ )
  {
    // Compute position in coordIndex list for this face
    i = depthsort ? ( faceIndex[face] << 2 ) : ( face << 2 );

    // Compute sequential index of first vertex
    int vert = depthsort ? ( faceIndex[face] * 3 ) : ( face * 3 );
```

```
      // Determine whether triangle is facing viewpoint
      SbBool facing = isFacing( coord->get3( coordIndex[i] ),
                                coord->get3( coordIndex[i+1] ),
                                coord->get3( coordIndex[i+2] ),
                                matrix, cosine );

      // Set depth-buffer function accordingly
      if ( facing && overlap )
        glDepthFunc( GL_ALWAYS );
      else
        glDepthFunc( GL_LEQUAL );

      glBegin( GL_POLYGON );

      // Do material change if required
      if ( matbind == SoMaterialBindingElement::PER_FACE )
        lazyElt->sendDiffuseByIndex( face );
      else if ( matbind == SoMaterialBindingElement::PER_FACE_INDEXED )
        lazyElt->sendDiffuseByIndex( materialIndex[ face ] );

      // Send three vertices to OpenGL pipeline
      // (with normal vector and texture coords)
      for ( j = 0; j < 3; j++ )
        {
          int index = coordIndex[ i + j ];

          // Send normal vector
          if ( sendNormals )
            glNormal3fv( normals[vert].getValue() );

          // Send texture coordinate
          if ( doTextures )
            {
              if ( texbind == SoTextureCoordinateBindingElement::PER_VERTEX )
                glTexCoord2fv( tcoord->get2( vert ).getValue() );
              else if ( texbind ==
                SoTextureCoordinateBindingElement::PER_VERTEX_INDEXED )
                {
                  // Use either coordinate index or
                  // specified texture coordinate index
                  if ( texUseCoordIndex )
                    glTexCoord2fv(
                      tcoord->get2( index ).getValue() );
                  else
                    glTexCoord2fv(
                      tcoord->get2( textureCoordIndex[vert] ).getValue() );
                }
            }

          // Send vertex position
          glVertex3fv( coord->get3( index ).getValue() );

          vert++;
        }

      glEnd();
    }

  lazyElt->reset( state, SoLazyElement::DIFFUSE_MASK );

  // Restore OpenGL depth-buffer function
  glDepthFunc( GL_LEQUAL );

  // Disable stencil operations
  glDisable( GL_STENCIL_TEST );

  // Free up memory
  free( (void *) normals );
  if ( depthsort )
    {
      delete faceIndex;
      delete faceDistance;
    }
}
```

```
// Method to determine whether triangle is facing viewing direction or not.
//
// p0, p1, p2 : vertices of triangle
// matrix : overall geometrical transformation acting on triangle w.r.t. viewpoint;
//          applying this matrix to a point or vector transforms it into the local
//          coordinate system of the observer (i.e. viewpoint is origin, z-axis is
//          direction of viewing, y-axis is upward, etc.)
// cosine : cosine of angle beyond which which triangle is
//          considered to face away from viewpoint
//
SbBool ClothingLayer::isFacing( const SbVec3f& p0,
                                const SbVec3f& p1,
                                const SbVec3f& p2,
                                const SbMatrix& matrix, float cosine )
{
  // Compute normal vector
  SbVec3f normal = (p1-p0).cross(p2-p0);

  // Compute centre position
  SbVec3f centre = ((p0+p1+p2)/3.0);

  // Apply transformation matrix to normal
  matrix.multDirMatrix( normal, normal );
  normal.normalize();

  // Apply transformation matrix to centre
  matrix.multVecMatrix( centre, centre );
  centre.normalize();

  // Determine whether facing toward viewpoint or not
  return ( normal.dot( centre ) < -cosine );
}


// Quicksort array between specified points (in order of decreasing value)
//
// index : array of indices which will be sorted
// value : array of f.p. values by which the indices will be sorted
// a, b : start and end points in arrays between which to sort
//
void ClothingLayer::quicksort( int *index, float *value, int a, int b )
{
  int a0 = a, b0 = b, dir = 0;

  while ( b > a )
    {
      if ( value[a] < value[b] )
        {
          // Swap array entries
          float temp1 = value[a];
          value[a] = value[b];
          value[b] = temp1;
          int temp2 = index[a];
          index[a] = index[b];
          index[b] = temp2;

          dir = !dir;
        }
      if ( dir )
        a++;
      else
        b--;
    }

  if ( a > a0 ) quicksort( index, value, a0, a-1 );
  if ( b < b0 ) quicksort( index, value, b+1, b0 );
}
```

FILE: DepthSortedGroup.h

```
/*****************************************************************************/
/* DepthSortedGroup class header.                                            */
/*                                                                           */
/* Extends Group node, but renders its children in order according to        */
/* their distance from the current viewpoint.                                */
/*                                                                           */
/* James Anderson (c) 1995-1998                                              */
/*****************************************************************************/


#ifndef _DEPTHSORTEDGROUP_H_
#define _DEPTHSORTEDGROUP_H_


#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/fields/SoMFVec3f.h>
#include <Inventor/fields/SoMFFloat.h>
#include <Inventor/fields/SoMFInt32.h>
#include <Inventor/actions/SoActions.h>


// Class definition
//
class DepthSortedGroup : public SoGroup
{

  SO_NODE_HEADER( DepthSortedGroup );

public:

  // Initializes the class
  static void initClass();

  // Constructors
  DepthSortedGroup();
  DepthSortedGroup( int numChildren );

  // Method to recompute centre points of children
  void recomputeCentres( const SbViewportRegion& vport );

  // Add child (only Separator nodes allowed)
  inline void addChild( SoNode *child )
    {
      if ( child->isOfType( SoSeparator::getClassTypeId() ) )
        SoGroup::addChild( child );
    }

  // Insert child (only Separator nodes allowed)
  inline void insertChild( SoNode *child, int index )
    {
      if ( child->isOfType( SoSeparator::getClassTypeId() ) )
        SoGroup::insertChild( child, index );
    }

protected:

  // Generic traversal of children for any action
  virtual void doAction( SoAction *action );

  // Methods for specific actions
  virtual void getBoundingBox( SoGetBoundingBoxAction *action );
  virtual void GLRender( SoGLRenderAction *action );
  virtual void handleEvent( SoHandleEventAction *action );
  virtual void pick( SoPickAction *action );
  virtual void getMatrix( SoGetMatrixAction *action );
  virtual void search( SoSearchAction *action );
  virtual void write( SoWriteAction *action );

private:

  // Destructor
  virtual ~DepthSortedGroup();
```

```
  // List of centre points
  SoMFVec3f centres;

  // Used for ordering children according to distance
  SoMFFloat childDistances;
  SoMFInt32 childIndices;

};


#endif // _DEPTHSORTEDGROUP_H_
```

FILE: DepthSortedGroup.cpp

```
/*****************************************************************************/
/* DepthSortedGroup class body.                                              */
/*                                                                           */
/* Extends Group node, but renders its children in order according to        */
/* their distance from the current viewpoint.                                */
/*                                                                           */
/* James Anderson (c) 1995-1998                                              */
/*****************************************************************************/


#include <Inventor/misc/SoChildList.h>
#include <Inventor/elements/SoViewingMatrixElement.h>
#include <Inventor/elements/SoModelMatrixElement.h>

#include <iostream.h>

#include "DepthSortedGroup.h"


SO_NODE_SOURCE( DepthSortedGroup );


// Static method to initialize class
//
void DepthSortedGroup::initClass()
{
  SO_NODE_INIT_CLASS( DepthSortedGroup, SoGroup, "Group" );
}


// Constructor
//
DepthSortedGroup::DepthSortedGroup()
{
  SO_NODE_CONSTRUCTOR( DepthSortedGroup );
}


// Other constructor
//
DepthSortedGroup::DepthSortedGroup( int numChildren ) : SoGroup( numChildren )
{
  SO_NODE_CONSTRUCTOR( DepthSortedGroup );
}


// Destructor
//
DepthSortedGroup::~DepthSortedGroup()
{
}


// The following methods for handling actions proceed
// in the same way as for the Group node
//

void DepthSortedGroup::getBoundingBox( SoGetBoundingBoxAction *action )
{
  SoGroup::getBoundingBox( action );
}
```

```
void DepthSortedGroup::handleEvent( SoHandleEventAction *action )
{
  SoGroup::handleEvent( action );
}

void DepthSortedGroup::pick( SoPickAction *action )
{
  SoGroup::pick( action );
}

void DepthSortedGroup::write( SoWriteAction *action )
{
  SoGroup::write( action );
}

void DepthSortedGroup::getMatrix( SoGetMatrixAction *action )
{
  SoGroup::getMatrix( action );
}

void DepthSortedGroup::search( SoSearchAction *action )
{
  SoGroup::search( action );
}


// Customized method for GL rendering
//
void DepthSortedGroup::GLRender( SoGLRenderAction *action )
{
  int i, j;

  // SoAction has a method called "getPathCode()" that returns
  // a code indicating how this node is related to the path(s)
  // the action is being applied to. This code is one of the
  // following:
  //
  // NO_PATH    = Not traversing a path (action was applied
  //                to a node)
  // IN_PATH    = This node is in the path chain, but is not
  //                the tail node
  // BELOW_PATH = This node is the tail of the path chain or
  //                is below the tail
  // OFF_PATH   = This node is off to the left of some node in
  //                the path chain
  //
  // If getPathCode() returns IN_PATH, it returns (in its two
  // arguments) the indices of the next nodes in the paths.
  // (Remember that an action can be applied to a list of
  // paths.)

  // For the IN_PATH case, these will be set by getPathCode()
  // to contain the number of child nodes that are in paths and
  // the indices of those children, respectively. In the other
  // cases, they are not meaningful.

  int         numIndices;
  const int   *indices;

  // Determine whether this node is in a path.
  SbBool inPath = action->getPathCode( numIndices, indices ) == SoAction::IN_PATH;

  if ( inPath )
    // In a path, so traverse only specific children
    for ( i = 0; i < numIndices; i++ )
      children->traverse( action, indices[i] );
  else
    {
      // Compute distances of children from
      // viewpoint and hence order of traversal

      childDistances.deleteValues( 0 );
      childIndices.deleteValues( 0 );

      // Get traversal state
      SoState *state = action->getState();

      // Get current viewing matrix
```

```
    SbMatrix vmatrix = SoViewingMatrixElement::get( state );

    // Get cumulative matrix
    SbMatrix matrix = SoModelMatrixElement::get( state );

    // Compute distances
    for ( i = 0; i < getNumChildren(); i++ )
      {
        SbVec3f point = centres[i];

        // Transform point
        matrix.multVecMatrix( point, point );
        vmatrix.multVecMatrix( point, point );

        // Compute distance
        float dist = point.length();

        // Place point in list, ordered by decreasing distance
        j = 0;
        while ( j < i && dist <= childDistances[j] )
          j++;
        childDistances.insertSpace( j, 1 );
        childDistances.set1Value( j, dist );
        childIndices.insertSpace( j, 1 );
        childIndices.set1Value( j, i );
      }

    // Traverse in order
    for ( i = 0; i < getNumChildren(); i++ )
      children->traverse( action, childIndices[i] );
  }
}


// Generic traversal of children for any action
//
void DepthSortedGroup::doAction( SoAction *action )
{
  SoGroup::doAction( action );
}


// Method to recompute centre points of children
//
void DepthSortedGroup::recomputeCentres( const SbViewportRegion& vport )
{
  // Clear list of centres
  centres.deleteValues( 0 );

  // Compute centre point of each child
  SoGetBoundingBoxAction gbba( vport );
  for ( int i = 0; i < getNumChildren(); i++ )
    {
      // Apply GetBoundingBox action to subgraph
      gbba.apply( getChild( i ) );

      centres.set1Value( i, gbba.getCenter() );
    }
}
```

FILE: StencilClear.h

```
/*************************************************************/
/* StencilClear class header.                                */
/*                                                           */
/* This node does nothing but clear the GL stencil buffer */
/* when traversed during rendering.                          */
/*                                                           */
/* James Anderson (c) 1995-1998                              */
/*************************************************************/


#ifndef _STENCILCLEAR_H_
#define _STENCILCLEAR_H_


#include <Inventor/nodes/SoNode.h>
#include <Inventor/fields/SoSFBool.h>
#include <Inventor/actions/SoActions.h>


// Class definition
//
class StencilClear : public SoNode
{

  SO_NODE_HEADER( StencilClear );

public:

  // Field
  SoSFBool clear;

  // Initializes the class
  static void initClass();

  // Constructors
  StencilClear();

protected:

  // Generic traversal for any action
  virtual void doAction( SoAction *action );

  // Methods for specific actions
  virtual void GLRender( SoGLRenderAction *action );
  virtual void getBoundingBox( SoGetBoundingBoxAction *action );
  virtual void handleEvent( SoHandleEventAction *action );
  virtual void pick( SoPickAction *action );
  virtual void getMatrix( SoGetMatrixAction *action );
  virtual void search( SoSearchAction *action );
  virtual void write( SoWriteAction *action );

private:

  // Destructor
  virtual ~StencilClear();

};


#endif // _STENCILCLEAR_H_
```

FILE: StencilClear.cpp

```
/*************************************************************/
/* StencilClear class header.                                */
/*                                                           */
/* This node does nothing but clear the GL stencil buffer */
/* when traversed during rendering.                          */
/*                                                           */
/* James Anderson (c) 1995-1998                              */
/*************************************************************/
```

```
#include "StencilClear.h"

SO_NODE_SOURCE( StencilClear );

// Static method to initialize class
//
void StencilClear::initClass()
{
  SO_NODE_INIT_CLASS( StencilClear, SoNode, "Node" );
}

// Constructor
//
StencilClear::StencilClear()
{
  SO_NODE_CONSTRUCTOR( StencilClear );

  SO_NODE_ADD_FIELD( clear, (TRUE) );
}

// Destructor
//
StencilClear::~StencilClear()
{
}

// The following methods for handling actions proceed
// in the same way as for the basic Node
//

void StencilClear::doAction( SoAction *action )
{
  SoNode::doAction( action );
}

void StencilClear::getBoundingBox( SoGetBoundingBoxAction *action )
{
  SoNode::getBoundingBox( action );
}

void StencilClear::handleEvent( SoHandleEventAction *action )
{
  SoNode::handleEvent( action );
}

void StencilClear::pick( SoPickAction *action )
{
  SoNode::pick( action );
}

void StencilClear::getMatrix( SoGetMatrixAction *action )
{
  SoNode::getMatrix( action );
}

void StencilClear::search( SoSearchAction *action )
{
  SoNode::search( action );
}

void StencilClear::write( SoWriteAction *action )
{
  SoNode::write( action );
}
```

```
// Customized method for GL rendering
//
void StencilClear::GLRender( SoGLRenderAction *action )
{
  // If field is TRUE, clear GL stencil buffer
  if ( clear.getValue() )
    glClear( GL_STENCIL_BUFFER_BIT );
}
```

FILE: StencilSeparator.h

```
/*************************************************************/
/* StencilSeparator class header.                            */
/*                                                           */
/* Extends Separator node, setting values in                 */
/* GL stencil buffer where children are drawn.               */
/*                                                           */
/* James Anderson (c) 1995-1998                              */
/*************************************************************/


#ifndef _STENCILSEPARATOR_H_
#define _STENCILSEPARATOR_H_


#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/actions/SoActions.h>


// Class definition
//
class StencilSeparator : public SoSeparator
{

  SO_NODE_HEADER( StencilSeparator );

public:

  // Initializes the class
  static void initClass();

  // Constructors
  StencilSeparator();
  StencilSeparator( int numChildren );

protected:

  // Generic traversal of children for any action
  virtual void doAction( SoAction *action );

  // Methods for specific actions
  virtual void GLRender( SoGLRenderAction *action );
  virtual void getBoundingBox( SoGetBoundingBoxAction *action );
  virtual void handleEvent( SoHandleEventAction *action );
  virtual void pick( SoPickAction *action );
  virtual void getMatrix( SoGetMatrixAction *action );
  virtual void search( SoSearchAction *action );
  virtual void write( SoWriteAction *action );
  // These must be overridden too...
  virtual void GLRenderBelowPath( SoGLRenderAction *action );
  virtual void GLRenderInPath( SoGLRenderAction *action );
  virtual void GLRenderOffPath( SoGLRenderAction *action );

private:

  // Destructor
  virtual ~StencilSeparator();

  // Convenience method to enable writing to stencil buffer
  inline void stencilOn()
    {
      // Set up stencil function and operation
      glStencilFunc( GL_ALWAYS, 1, 1 );
      glStencilOp( GL_KEEP, GL_KEEP, GL_REPLACE );
      glEnable( GL_STENCIL_TEST );
    }
```

```
  // Convenience method to disable writing to stencil buffer
  inline void stencilOff()
    {
      // Restore stencil operation
      glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
      glDisable( GL_STENCIL_TEST );
    }

};

#endif // _STENCILSEPARATOR_H_
```

FILE: StencilSeparator.cpp

```
/*************************************************************/
/* StencilSeparator class body.                              */
/*                                                           */
/* Extends Separator node, setting values in                 */
/* GL stencil buffer where children are drawn.               */
/*                                                           */
/* James Anderson (c) 1995-1998                              */
/*************************************************************/


#include <Inventor/misc/SoChildList.h>

#include "StencilSeparator.h"


SO_NODE_SOURCE( StencilSeparator );


// Static method to initialize class
//
void StencilSeparator::initClass()
{
  SO_NODE_INIT_CLASS( StencilSeparator, SoSeparator, "Separator" );
}


// Constructor
//
StencilSeparator::StencilSeparator()
{
  SO_NODE_CONSTRUCTOR( StencilSeparator );
}


// Other constructor
//
StencilSeparator::StencilSeparator( int numChildren ) : SoSeparator( numChildren )
{
  SO_NODE_CONSTRUCTOR( StencilSeparator );
}


// Destructor
//
StencilSeparator::~StencilSeparator()
{
}


// The following methods for handling actions proceed
// in the same way as for the Separator node
//

void StencilSeparator::doAction( SoAction *action )
{
  SoSeparator::doAction( action );
}
```

```
void StencilSeparator::getBoundingBox( SoGetBoundingBoxAction *action )
{
  SoSeparator::getBoundingBox( action );
}

void StencilSeparator::handleEvent( SoHandleEventAction *action )
{
  SoSeparator::handleEvent( action );
}

void StencilSeparator::pick( SoPickAction *action )
{
  SoSeparator::pick( action );
}

void StencilSeparator::getMatrix( SoGetMatrixAction *action )
{
  SoSeparator::getMatrix( action );
}

void StencilSeparator::search( SoSearchAction *action )
{
  SoSeparator::search( action );
}

void StencilSeparator::write( SoWriteAction *action )
{
  SoSeparator::write( action );
}


// Customized methods for GL rendering
//

void StencilSeparator::GLRenderBelowPath( SoGLRenderAction *action )
{
  stencilOn();

  // Call parent method to render
  SoSeparator::GLRenderBelowPath( action );

  stencilOff();
}

void StencilSeparator::GLRenderInPath( SoGLRenderAction *action )
{
  stencilOn();

  // Call parent method to render
  SoSeparator::GLRenderInPath( action );

  stencilOff();
}

void StencilSeparator::GLRenderOffPath( SoGLRenderAction *action )
{
  stencilOn();

  // Call parent method to render
  SoSeparator::GLRenderOffPath( action );

  stencilOff();
}

void StencilSeparator::GLRender( SoGLRenderAction *action )
{
  stencilOn();

  // Call parent method to render
  SoSeparator::GLRender( action );

  stencilOff();
}
```

# Appendix D

# The Likert Questionnaire Format

This appendix provides a brief account of the Likert questionnaire format and its employment within the usability trial described in Chapter 7.

Questionnaires and attitude scales are instruments for gathering structured information from people. It is important for any usability experiment, if attitude questionnaires are to be used, that they provide a reliable and valid measuring instrument for testing a particular opinion or pattern of behaviour. In order to assess the subjective elements of the usability trial described in this thesis, a Likert-type attitude scale (Likert, 1932) was employed within the questionnaires completed by subjects. The advantages of using the Likert technique have been identified by Coolican (1994) as:

1. Subjects prefer the Likert scaling technique because it is "more natural" to complete and because it maintains the subject's direct involvement.
2. The Likert technique has been shown to have a high degree of validity and reliability.
3. The Likert scale has been shown to be effective at measuring changes over time.

In addition, the equal-interval characteristic of the Likert format permits the use of parametric statistical analysis (*e.g.* mean user attitude for the subject sample) of the questionnaire results.

The questionnaires employed in the usability trial consisted of a set of attitude stimulus statements with which the subjects indicated their level of agreement using

a seven-point scale with a mid-neutral point (see Appendix E). For each statement, the overall attitude for each statement is obtained by taking the mean score (from 1 to 7) across the subject sample after correcting for positively-worded or negatively-worded statements where appropriate.

In designing a Likert-type attitude questionnaire, there are several important issues which must be addressed. Firstly, and most importantly, it is necessary to have the stimulus statements cover all the relevant aspects of usability under investigation in the test. In the present case, these include such aspects as efficiency, accuracy, satisfaction, visual appearance, and comparison to alternatives. Secondly, it is essential to achieve a balance between positive and negative attitude statements; this overcomes the danger of the overall score obtained from the attitude scale reflecting the users' biased tendency to agree rather than disagree with the questionnaire statements (an effect known as 'response acquiescence set') instead of providing valid information on their attitudes. The effect in question is usually due to the fact that the agree-disagree scale consistently goes from left to right and there is a tendency for subjects to fill in only one 'position' on the page. Finally, attention must be paid to well-known phenomena such as the 'halo effect' (in which subjects let themselves be influenced in their response to individual statements by an overall feeling towards the task) and the problem of central tendency (in which subjects tend to choose the mid-points on scales since this does not commit them to any strong opinion on the issue one way or the other) by carefully considering both the ordering of the statements and the 'strength' of their wording (aiming to ensure that potential responses to a statement will span the full range of the Likert scale).

# Appendix E

# Experiment Questionnaires

The questionnaire on the following page was completed by participants in the user trial (see Chapter 7) after using each of the two fitting rooms.

**Please complete this questionnaire for the virtual fitting room you have just used. For each statement below, tick the box which best expresses your opinion on that statement.**

| | | Strongly Agree | Agree | Slightly Agree | Neither Agree nor Disagree | Slightly Disagree | Disagree | Strongly Disagree |
|---|---|---|---|---|---|---|---|---|
| *Q1* | *The mannequin accurately showed how the clothes would fit me.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q2* | *It took too long for the clothing to settle onto the mannequin.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q3* | *The clothing looked realistic when modelled on the mannequin.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q4* | *I enjoyed using the fitting room.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q5* | *I felt bored when using the fitting room.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q6* | *I would be reluctant to buy clothes having tried them on in a virtual fitting room like this.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q7* | *The mannequin showed how the clothes would suit me.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q8* | *I would prefer to see the clothing modelled in a catalogue.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q9* | *I would prefer to try the clothing myself in a real fitting room.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q10* | *I was satisfied with what I saw in the fitting room.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q11* | *I liked the appearance of the mannequin dressed with the clothing.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q12* | *I did not find the fitting room helpful.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q13* | *I think that the fitting room needs to be improved.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| *Q14* | *I like the idea of a virtual fitting room.* | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

The experiment supervisor completed the following questionnaire by interviewing each participant at the end of the session (and having chosen between the two fitting rooms).

---

The subject chose the _____ fitting room, because… *(tick all that apply)*

…it was more helpful [____]

…it was more efficient [____]

…the virtual mannequin looked more realistic [____]

…the clothing looked more realistic [____]

…it took less time to dress the mannequin [____]

…it was more enjoyable [____]

Other reason *(give details below)* [____]

_____

_____

_____

_____

_____

_____

---

# Appendix F

# Publications

**Anderson, J. and Jack, M. (1996)**, *Dressing the Virtual Teleshoppe User*, Proceedings of the European Conference on Multimedia Applications, Services and Techniques (ECMAST'96), pp. 293-303.

**Anderson, J. and Jack, M. (1997)**, *A Fast Collision Detection Technique for Modelling Virtual Clothing*, Proceedings of the Eurographics UK Chapter 15th Annual Conference, pp. 75-88.

**Anderson, J. and Jack, M. (1997)**, *A Virtual Mannequin Service Using the FIGMENT Scheme*, International Journal of Virtual Reality, vol. 3, no. 2, pp. 1-11.

**Anderson, J. and Jack, M. (1998)**, *FIGMENT: A Basis for Interactive Virtual Mannequin Services*, Proceedings of the 1998 Workshop on Multimedia Signal Processing, IEEE Signal Processing Society (forthcoming).

# References

**Algorri, M. and Schmitt, F. (1996)**, *Mesh Simplification*, Computer Graphics Forum (proc. EUROGRAPHICS'96), vol. 15, no. 3, pp. 77-86.

**Bouknight, W.J. (1970)**, *A Procedure for Generation of Three-Dimensional Half-Toned Computer Graphics Presentations*, Communications of the ACM, vol. 13, no. 9, pp. 527-536.

**Breen, D.E., House, D.H. and Wozny, M.J. (1994a)**, *Predicting the Drape of Woven Cloth using Interacting Particles*, Computer Graphics (proc. SIGGRAPH'94), pp. 365-372.

**Breen, D.E., House, D.H. and Wozny, M.J. (1994b)**, *A Particle-Based Model for Simulation the Draping Behavior of Woven Cloth*, Textile Research Journal, vol. 64, no. 11, pp. 663-685.

**Carignan, M., Yang, Y., Magnenat Thalmann, N. and Thalmann, D. (1992)**, *Dressing Animated Synthetic Actors with Complex Deformable Clothes*, Computer Graphics (proc. SIGGRAPH'92), pp. 99-104.

**Catmull, E. (1974)**, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, Ph.D. thesis, Computer Science Department, University of Utah, Salt Lake City.

**Chadwick, J., Haumann, D. and Parent, R. (1989)**, *Layered Construction for Deformable Animated Characters*, Computer Graphics, vol. 23, no. 3, pp. 243-252.

**Cohen, J., Lin, M., Manocha, D. and Ponamgi, M. (1995)**, *I-collide: An Interactive and Exact Collision Detection System for Large-Scale Environments*, Proceedings of ACM Interactive 3D Graphics Conference, pp. 112-120.

**Cohen, J., Varshney, A., Manocha, D., Turk, G., Weber, H., Agarwal, P., Brooks, F. and Wright, W. (1996)**, *Simplification Envelopes*, Computer Graphics (proc. SIGGRAPH'96), pp. 119-128.

**Coolican, H. (1994),** *Research Methods and Statistics in Psychology*, Hodder & Stoughton.

**Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M. and Stuetzle, W. (1995)**, *Multiresolution Analysis of Arbitrary Meshes*, Computer Graphics (proc. SIGGRAPH'95), pp. 173-182.

**Goldberg, D. E. (1989)**, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Mass.

**Hannah, J. and Hillier, M.J. (1995)**, *Applied Mechanics ($3^{rd}$ edition)*, Longman Publishing Group, pp. 37-63.

**Hibbeler, R.C. (1994)**, *Mechanics of Materials ($2^{nd}$ edition)*, Macmillan College Publishing Company, New York, pp. 447-457.

**Hoppe, H., DeRose, T., Duchamp, T., McDonald, J. and Stuetzle, W. (1993)**, *Mesh Optimization,* Computer Graphics (proc. SIGGRAPH'93), pp. 19-26.

**Hoppe, H. (1996)**, *Progressive Meshes*, Computer Graphics (proc. SIGGRAPH'96), pp. 99-108.

**Hubbard, P. (1993)**, *Interactive Collision Detection*, Proceedings of the 1993 IEEE Symposium on Research Frontiers in Virtual Reality, pp. 24-31.

**Hubbard, P. (1995a)**, *Real-Time Collision Detection and Time-Critical Computing*, Workshop on Simulation and Interaction in Virtual Environments, pp. 92-96.

**Hubbard, P. (1995b)**, *Collision Detection for Interactive Graphics Applications*, IEEE Transactions on Visualisation and Computer Graphics, pp. 218-230.

**Hubbard, P. (1996)**, *Approximating Polyhedra with Spheres for Time-Critical Collision Detection*, ACM Transactions on Graphics, vol. 15, no. 3, pp. 179-210.

**Kawabata, S. (1980)**, *The Standardization and Analysis of Hand Evaluation*, The Textile Machinery Society of Japan, Osaka.

**Kalvin, A.D. and Taylor, R.H. (1996)**, *Superfaces: Polygonal Mesh Simplification with Bounded Error*, IEEE Computer Graphics and Applications, vol. 16, no. 3, pp. 64-77.

**Kempf, R. (ed.) and Frazier, C. (ed.) (1997)**, *OpenGL Reference Manual : The Official Reference Document to OpenGL, Version 1.1*, Addison-Wesley, Reading, Mass.

**Kreyszig, E. (1988)**, *Advanced Engineering Mathematics ($6^{th}$ edition)*, John Wiley & Sons, pp. 1062-1071.

**Krishnamurthy, V. and Levoy, M. (1996)**, *Fitting Smooth Surfaces to Dense Polygon Meshes*, Computer Graphics (proc. SIGGRAPH'96), pp. 313-324.

**Likert, R.A. (1932)**, *A Technique for the Measurement of Attitudes*, Archives of Psychology, vol. 140.

**Lin, M.C. (1993)**, *Efficient Collision Detection for Animation and Robotics*, Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley.

**Lin, M.C. and Manocha, D. (1995)**, *Efficient Contact Determination Between Geometric Models*, International Journal of Computational Geometry and Applications, vol. 7, no. 1 & 2, pp. 123-151.

**Louchet, J. (1994)**, *An Evolutionary Algorithm for Physical Motion Analysis*, Proceedings of the British Machine Vision Conference, York, UK.

**Louchet, J., Provot, X. and Crochemore, D. (1995)**, *Evolutionary Identification of Cloth Animation Models*, Eurographics Workshop on Animation and Simulation, pp. 44-54.

**Lounsbery, M. (1994)**, *Multiresolution Analysis for Surfaces of Arbitrary Topological Type*, Ph.D. thesis, Department of Computer Science and Engineering, University of Washington.

**Moore, M. and Wilhelms, J. (1988)**, *Collision Detection and Response for Computer Animation*, Computer Graphics (proc. SIGGRAPH'88), pp. 289-297.

**Ponamgi, M. K., Manocha, D. and Lin, M. C. (1997)**, *Incremental Algorithms for Collision Detection Between Solid Models,* IEEE Transactions on Visualization and Computer Graphics, vol. 3, no. 1, pp. 51-64.

**Provot, X. (1995)**, *Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior*, Proc. Graphics Interface '95, pp. 147-154.

**Ronfard, R. and Rossignac, J. (1996)**, *Full-Range Approximation of Triangulated Polyhedra*, Computer Graphics Forum (proc. EUROGRAPHICS'96), pp. 67-76.

**Schroeder, W., Zarge, J. and Lorensen, W. (1992)**, *Decimation of Triangle Meshes*, Computer Graphics (proc. SIGGRAPH'92), pp. 65-70.

**Terzopoulos, D. and Fleischer, K. (1988)**, *Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture*, Computer Graphics (proc. SIGGRAPH'88), pp. 269-278.

**Thibault, W.C. and Naylor, B.F. (1987)**, *Set Operations on Polyhedra Using Binary Space Partitioning Trees*, Computer Graphics (proc. SIGGRAPH'87), pp. 153-162.

**Turk, G. (1992)**, *Re-tiling of Polygonal Surfaces*, Computer Graphics (proc. SIGGRAPH'92), pp. 55-64.

**Volino, P., Courchesne, M. and Magnenat Thalmann, N. (1995)**, *Versatile and Efficient Techniques for Simulating Cloth and Other Deformable Objects*, Computer Graphics (proc. SIGGRAPH'95), pp. 137-144.

**Volino, P. and Magnenat Thalmann, N. (1994)**, *Efficient Self-Collision Detection on Smoothly Discretised Surface Animations using Geometrical Shape Regularity*, Computer Graphics Forum (proc. EUROGRAPHICS'94), pp. 155-166.

**Volino P., Magnenat Thalmann N., Jianhua S. and Thalmann D. (1996)**, *The Evolution of a 3D System for Simulating Deformable Clothes on Virtual Actors,* IEEE Computer Graphics and Applications, vol. 16, no. 5, pp. 42-50.

**Volino, P. and Magnenat Thalmann, N. (1997)**, *Developing Simulation Techniques for an Interactive Clothing System*, Proceedings of International Conference on Virtual Systems and Multimedia '97, Geneva, Switzerland, pp. 109-118.

**Von Herzen, B., Barr, A.H. and Zatz, H.R. (1990)**, *Geometric Collisions for Time-Dependent Parametric Surfaces*, Computer Graphics (proc. SIGGRAPH'90), pp. 39-48.

**Watkins, G.S. (1970)**, *A Real Time Visible Surface Algorithm*, Ph.D. thesis, Computer Science Department, University of Utah, Salt Lake City.

**Wernecke, J. (1994a)**, *The Inventor Mentor*, Addison-Wesley, Reading, Mass.

**Wernecke, J. (1994b)**, *The Inventor Toolmaker*, Addison-Wesley, Reading, Mass.

**Woo, M., Neider, J. and Davis, T. (1997)**, *OpenGL Programming Guide : The Official Guide to Learning OpenGL, Version 1.1*, Addison-Wesley, Reading, Mass.

**Wylie, C., Romney, G.W., Evans, D.C. and Erdahl, A.C. (1967)**, *Halftone Perspective Drawings by Computer*, Proceedings of the Fall Joint Computer Conference '67, pp. 49-58.

**Yang, Y. and Magnenat Thalmann, N. (1993)**, *An Improved Algorithm for Collision Detection in Cloth Animation with Human Body*, Proceedings of the First Pacific Conference on Computer Graphics and Applications, vol. 1, pp. 237-251.